RIPRENDIAMO I PUNTATORI

- Ogni variabile in C è una astrazione di una cella di memoria a cui corrisponde un nome, un contenuto e un indirizzo.

```
int a = 5;
```

a | 5 | α=&a

- Esistono in C particolari variabili dette **puntatori** che possono contenere un indirizzo di una variabile

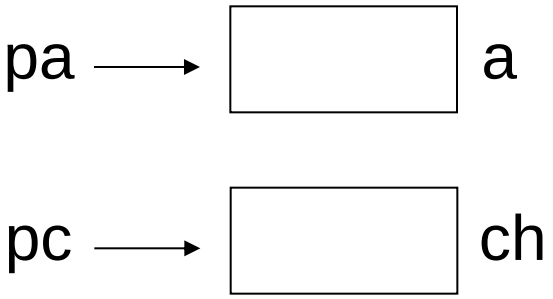
RIPRENDIAMO I PUNTATORI

- La sintassi è

  `tipoBase * varPunt;`

- dove **`varPunt`** è definita come variabile di tipo puntatore a **`tipoBase`**
- Quindi **`varPunt`** può contenere indirizzi di variabili di tipo **`tipoBase`** che può essere int, float, double o char

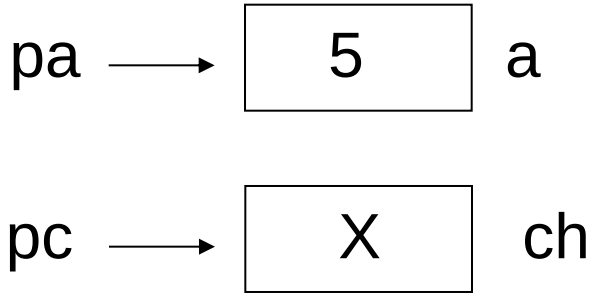
```
int a; char ch;
int *pa; char *pc;
pa = &a;
pc = &ch;
```



RIPRENDIAMO I PUNTATORI

- Ora se assegnamo valori ad a e c

```
int a; char ch;
int *pa; char *pc;
pa = &a;
pc = &ch;
a = 5;
ch = 'X';
```




```
printf("a=%d  ch =%c", a,ch);
printf("a=%d  ch =%c", *pa,*pc);
```

Hanno lo stesso effetto

ARRAY E PUNTATORI

- Sappiamo che il nome dell'array è l'indirizzo della prima cella di memoria

```
int buf[100];
int *punt;
punt = buf
```
oppure
```
punt = &buf[0]
```

sono equivalenti.

Da ora in poi possiamo usare punt per accedere all'array. COME?????

ARRAY E PUNTATORI

```
int buf[100];
int *punt;
```

`punt = buf` oppure **`punt = &buf[0]`**

sono equivalenti.

Da ora in poi possiamo usare punt per accedere all'array.

```
buf[7] = 5;
*(punt + 7) = 5
```

Hanno lo stesso effetto di assegnare 5 all'ottava cella del vettore

ARITMETICA DEI PUNTATORI

- Per una variabile di tipo puntatore esistono operazioni aritmetiche: l'incremento, il decremento, la somma, la sottrazione ecc…
- Ma qual è l'esatto significato di tali operazioni?
- Se ho una variabile di tipo puntatore a carattere

```
char *pc;
```

- e **`pc`** vale 10 (indirizzo == 10) non è detto che **`pc++`** valga 11. Tutto dipende dal tipoBase puntato

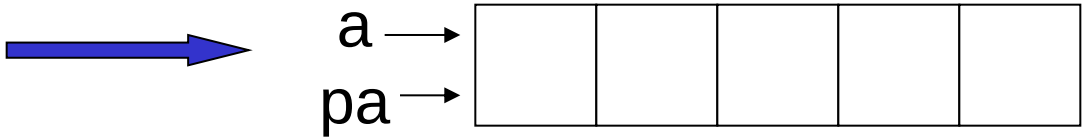
ARITMETICA DEI PUNTATORI

- Incrementare un puntatore di uno significa far saltare il puntatore alla prossima locazione corrispondente all'elemento di memoria il cui tipo corrisponde al tipoBase

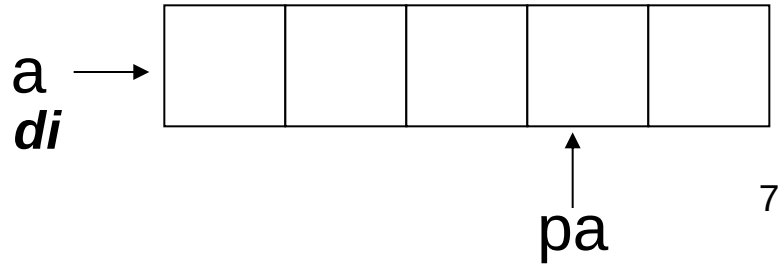
```
int a[5];

int* pa;

pa = a
```



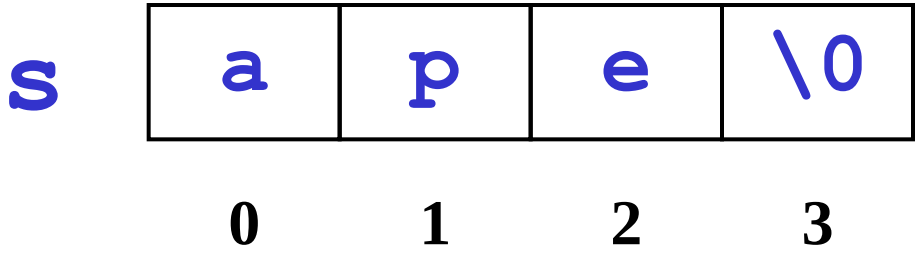

```
pa  = pa + 3;
```



Corrisponde allo spostamento di pa di 3 posizioni dove <u>ogni posizione occupa lo spazio di un int</u>

STRINGHE: ARRAY DI CARATTERI

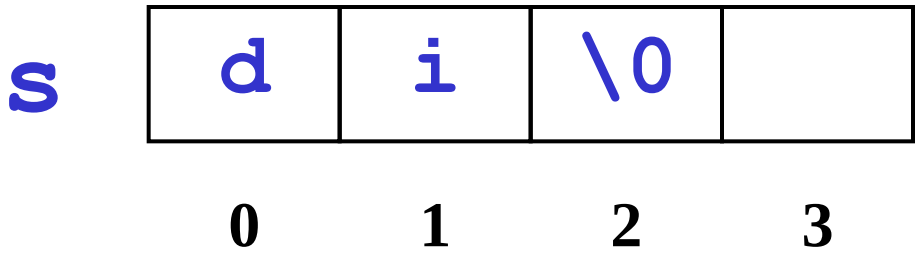
- Una *stringa di caratteri in C* è un array di caratteri *terminato dal carattere* `'\0'`



- Un vettore di N caratteri può dunque ospitare stringhe *lunghe al più N-1 caratteri,* perché una cella è destinata al terminatore `'\0'`.

STRINGHE: ARRAY DI CARATTERI

- Un array di N caratteri può ben essere usato per memorizzare *stringhe più corte*



- In questo caso, *le celle oltre la k-esima* (k essendo la lunghezza della stringa) *sono concettualmente vuote:* praticamente sono inutilizzate e contengono un valore casuale.

STRINGHE

- Una stringa si può *inizializzare,* come ogni altro array, elencando le singole componenti:

```
char s[4] = {'a', 'p', 'e', '\0'};
```

oppure anche, più brevemente, *con la forma compatta* seguente:

```
char s[4] = "ape" ;
```

Il carattere di terminazione '\0' è *automaticamente incluso* in fondo. *Attenzione alla lunghezza!*

STRINGHE: LETTURA E SCRITTURA

- Una stringa si può *leggere da tastiera e stampare,* come ogni altro array, elencando le singole componenti:

```
...char str[4]; int i;
  for (i=0; i < 3; i++)
  scanf("%c", &str[i]); str[3] = '\0' ...
```

- oppure anche, più brevemente, *con la forma compatta* seguente:

```
...char str[4]; scanf("%s", str);
```

NOTA: E' un'eccezione !!! Gli array non si possono leggere e scrivere interamente, ma elemento per elemento.

STRINGHE: LETTURA E SCRITTURA

- Si noti che usando la scanf

```
scanf("%s", str);
```

- La stringa viene considerata terminata anche dal carattere spazio bianco. Quindi se vogliamo scrivere come stringa “Giovanni Neri” non possiamo usare la scanf cosi com’è, ma cambiando la stringa di formato come segue:

```
scanf("%[^\n]", str);
```

- Altrimenti possiamo usare la gets

LETTURA E SCRITTURA DI STRINGHE

```
char *gets(char *s);
```

La funzione gets() legge una linea dallo stdin fino al carattere di new-line '\n' o di EOF. Questi caratteri sono sostituiti con '\0'.

Non viene eseguito nessun controllo sulla dimensione di s, pertanto se la linea da leggere supera la dimensione di s si ha un buffer overrun.

La funzione gets() ritorna s in caso di successo e NULL quando una fine file viene incontrata senza che nessun carattere sia stato letto.

```
int puts(const char *s);
```

La funzione puts() scrive la stringa s sullo stdout aggiungendo il carattere di new-line '\n'.

Restituisce un numero non-negativo in caso di successo e EOF in caso di errore.

ESEMPIO

<u>Problema:</u>

Date due stringhe di caratteri, decidere quale precede l'altra in ordine alfabetico.

<u>Rappresentazione dell'informazione:</u>

- poiché vi possono essere *tre* risultati (s1<s2, s1==s2, s2<s1), *un boolean non basta*
- possiamo usare:
  - due boolean (**`uguale`** e **`precede`**)
  - tre boolean (**`uguale`**, **`s1precedes2`**, **`s2precedes1`**)
  - un intero (negativo, zero, positivo)

scegliamo la terza via.

ESEMPIO

<u>Specifica:</u>

- scandire uno a uno gli elementi *di egual posizione* delle due stringhe, *o fino alla fine delle stringhe, o fino a che se ne trovano due diversi*
  - *nel primo caso, le stringhe sono uguali*
  - *nel secondo, sono diverse*
- nel secondo caso, confrontare i due caratteri così trovati, e determinare qual è il minore
  - la stringa a cui appartiene tale carattere precede l'altra

ESEMPIO

Codifica:

```
int main() {
 char s1[] = ”Maria";
 char s2[] = ”Marta";
 int i=0, stato;
 while(s1[i]!='\0' && s2[i]!='\0' &&
       s1[i]==s2[i]) i++;
 stato = s1[i]-s2[i];
 ……
return 0;
 }
```

negativo ↔ s1 precede s2
positivo ↔ s2 precede s1
zero ↔ s1 è uguale a s2

ESEMPIO

Problema:

Data una stringa di caratteri, copiarla in un altro array di caratteri (di lunghezza non inferiore).

Ipotesi:

La stringa è “ben formata”, ossia correttamente terminata dal carattere ‘\0’.

Specifica:

- scandire la stringa elemento per elemento, fino a trovare il terminatore ‘\0’ (che esiste certamente)
- *nel fare ciò, copiare l’elemento nella posizione corrispondente dell’altro array.*

ESEMPIO

<u>Codifica</u>:copia della stringa carattere per carattere

```
int main() {
 char s[] = "Nel mezzo del cammin di";
 char s2[40];
 int i;
 for (i=0; s[i]!='\0'; i++)
     s2[i] = s[i];
 s2[i] = '\0';
return 0;
}
```

La dimensione deve essere tale da garantire che la stringa non ecceda

Al termine, occorre garantire che anche la nuova stringa sia “ben formata”, inserendo esplicitamente il terminatore.

ESEMPIO

Perché non fare così?

```
int main() {
 char s[] = "Nel mezzo del cammin di";
 char s2[40];
     s2 = s;
…}
```

ERRORE DI COMPILAZIONE:
incompatible types in assignment !!

PERCHÉ GLI ARRAY NON POSSONO ESSERE MANIPOLATI NELLA LORO INTEREZZA !

ESEMPIO

Problema:

Data una stringa di caratteri, *scrivere una funzione* che ne calcoli la lunghezza.

Nel caso delle stringhe, la dimensione non serve
perché può essere dedotta dalla posizione dello '\0'

Codifica:

```
int lunghezza(char s[]) {
  int lung=0;
  for (lung=0; s[lung]!='\0'; lung++);
  return lung;
}
```

LIBRERIA SULLE STRINGHE

Il C fornisce una nutrita libreria di funzioni per operare sulle stringhe:

```
#include < string.h >
```

Include funzioni per:

- confrontare due stringhe (**`strcmp`**)
- copiare una stringa in un'altra (**`strcpy`**)
- calcolare la lunghezza di una stringa (**`strlen`**)
- concatenare due stringhe (**`strcat`**)
- cercare un carattere in una stringa (**`strchr`**)
- ...

LIBRERIA SULLE STRINGHE

```
int strcmp(const char *s1, const char *s2);
```

La funzione **strcmp()** effettua il confronto fra le due stringhe **s1** e **s2**.
restituisce un intero minore, uguale o maggiore di 0 a seconda che **s1** sia rispettivamente minore, uguale o maggiore di **s2**.

```
char *strcpy(char *dest, const char *src);
```

- La funzione **strcpy()** copia la stringa **src** sull'area puntata da **dest** la quale deve essere sufficientemente ampia ad accogliere tutti i caratteri di **src** compreso il terminatore '\0'.
  Le aree **src** e **dest** non si devono sovrapporre.

  Restituisce un puntatore alla stringa di destinazione **dest.**

LIBRERIA SULLE STRINGHE

```
int strlen(const char *s);
```

La funzione strlen() calcola la lunghezza della stringa s, (inteso come il numero di caratteri dell'array puntato da s) escluso il terminatore '\0'.

La funzione strlen() ritorna il numero di caratteri in s.

LIBRERIA SULLE STRINGHE

```
char *strcat(char *dest, const char *src);
```

La funzione strcat() concatena la stringa src aggiungendola al termine della stringa dest.
La concatenazione avviene copiando la stringa src a partire dal terminatore della stringa desc. La copia di src e' comprensiva del suo terminatore.
L'area di memoria puntata da dest deve essere sufficientemente ampia da accogliere entrambe le stringhe ed il terminatore '\0', quindi dest deve essere dimensionata almeno per contenere strlen(dest)+strlen(src)+1 caratteri.
Le funzioni strcat() ritornano il puntatore alla stringa dest risultante.

LIBRERIA SULLE STRINGHE

```
char *strchr(const char *s, int c);
```

La funzione individua la prima occorrenza di c (convertito ad un carattere) nella stringa s.

Il valore di ritorno e' il puntatore alla prima occorrenza di c o un puntatore nullo se c non e' stato trovato.

ESEMPIO FUNZIONI DI LIBRERIA

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

int main()
{char s1[10];
 char s2[20];
 char *s3;
 int l1,l2,l3;
 printf("inserisci due stringhe \n");
 scanf("%s", s1);
 scanf("%s", s2);

 l1= strlen(s1);
 l2= strlen(s2);
```

ESEMPIO FUNZIONI DI LIBRERIA

```
 printf("lunghezza della prima stringa : %d \n",l1);
 printf("lunghezza della seconda stringa : %d \n",l2);

 s3=strcat(s1,s2);

 printf("terza stringa %s\n", s3);

 l3 = strcmp(s1,s2);

 if (l3 < 0)
   printf("%s prima di %s", s1,s2);
   else if (l3 == 0)
        printf("stringhe uguali");
        else printf("%s dopo di %s", s1,s2);

 return 0;

}
```

LETTURA E SCRITTURA SU STRINGA

In C c'è la possibilità di leggere e scrivere direttamente da e su stringa

```
int sprintf(char *str, const char *format, ...);
```

La funzione **`sprintf()`** è analoga alla funzione **`printf()`** ma nella prima i caratteri vengono scritti nell'area puntata da str. Naturalmente str deve essere ampia a sufficienza per contenere i caratteri in output più il terminatore '\0'.

```
int sscanf(const char *str, const char *format, ...);
```

La funzione **`sscanf()`** ha lo stesso comportamento della funzione **`scanf()`**, ma l'input avviene da **str**.