ARRAY

tipoelemento nome [numero elementi]

NON SI POSSONO USARE VARIABILI PER LE DIMENSIONI DEI VETTORI

STRUTTURA DATI COSTANTE

int vettoreinteri [20]; (gli elementi vanno da 0 a 19)

Per accedere al quindicesimo elemento:

... vettoreinteri **[**15**]** ...

N.B.: il compilatore **non** controlla la correttezza dell'indice, ovvero che sia:

0 <= indice < numero elementi

Unix: tentare di accedere ad un array con un indice non corretto provoca ***segmentation fault***!

Definizione di array multidimensionali

tipoelemento nome [numeroelementi1dimensione]
[numeroelementi2dimensione]
...
[numeroelementiNdimensione]

ESEMPIO DI DEFINIZIONE E USO

Vettore monodimensionale:

```
typedef int vettore [30];
vettore v;  int i, nontrovato = 1, chiave; ...
for  ( i = 0; i < 30 && v[i] > 0 ; i = i + 1)        v[i]= 10;
i = 0;
while  (i < 30 && v[i] > 0){  v[i] = 10;    i= i + 1;      }
 ...
for  ( i = 0; i < 30 && nontrovato ; i = i + 1)
        {if (v[i] == chiave) nontrovato = 0;}
```

Vettore bidimensionale (matrice):

```
int i, j, k;
int M1 [10][20], M2[20][30], M3[10][30];

for (i=0; i < 10; i=i+1)              /* inizializzazione di M3 */
     for (j=0; j < 30; j=j+1) M3[i][j] = 0;

                /* ripetizioni enumerative */


/* prodotto di matrici M1 e M2 in M3 */
for (i=0; i < 10; i=i+1)
     for (j=0; j < 20; j=j+1)
          for (k=0; k < 30; k=k+1)
M3[i][k] = M3[i][k] + M1[i][j] * M2[j][k];
```

ESEMPIO DI INIZIALIZZAZIONE

Vettore monodimensionale:

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
/*  v[0] = 1; v[1] = 2; ... v[9] = 10;  */

int v[ ] = {1,2,3,4,5,6,7,8,9,10};
```

Memorizzazione per righe
l'indice più a destra varia più velocemente

matrix

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3

Vettore bidimensionale (matrice):

```
int matrix[4][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};

int matrix[ ][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};
```


PUNTATORI

Definizione di una variabile puntatore

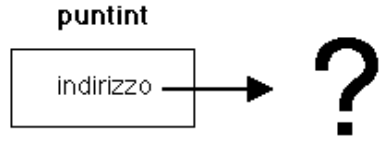
```
tipoElementoPuntato    *nomePuntatore
```

Esempio:

```
int  k,*puntint, x;        x = 5;
```

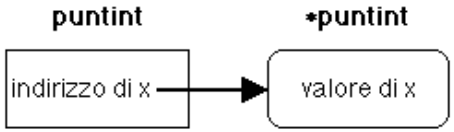
puntint è il puntatore ➠ contiene l'indirizzo dell'elemento puntato

*puntint è l'elemento puntato ➠ contiene il valore (intero) dell'elemento puntato



OPERATORE INDIRIZZO

```
puntint = &x;          /*  &x e' l'indirizzo di x  */
```




```
k = *puntint;                /*  k = 5  */
*puntint = k + 1;        /*  x = 6  */
```

N.B.: l'operatore indirizzo non ha alcun effetto se applicato ad un array, in quanto l'array è una struttura dati costante:

Es.

```
int vector[30];
vector = &vector = &&vector …
```

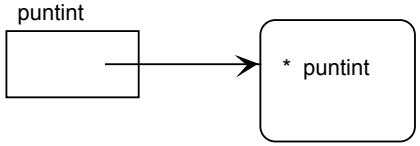
UGUAGLIANZA vs. **IDENTITÀ**

relazione sui puntati — relazione sui puntatori

ALLOCAZIONE DINAMICA

void * **malloc**(size_t size);
alloca un blocco di size byte in memoria centrale (nell'area **heap**)
restituisce l'**indirizzo del primo byte**, oppure **0** in caso di mancanza di spazio

void **free**(void *block);
dealloca un blocco di memoria di indirizzo iniziale block, allocato con una precedente chiamata alla malloc



non sono definite
a livello di **linguaggio di programmazione**
ma a livello di **sistema operativo**

Esempio:

```
int *p;
   ...  p non è ancora definito  ...
p = (int *) malloc(sizeof (int));
   ...  p è definito, il suo contenuto non ancora  ...
*p = 55;       /*  posso scrivere p[0] = 55 ?  */
   ...  p e *p sono definiti e utilizzabili  ...
free(p);
   ...  p non è più definito  ...
```


Problemi connessi con i puntatori

DANGLING REFERENCES

Possibilità di fare riferimento ad aree di memoria non più allocate al programma

```
int *p;
     p = (int *) malloc(sizeof(int));
     ...
     free(p);
     ...  *p  ...              /*  Da non fare!  */
```

AREE INUTILIZZABILI

Possibilità di perdere il riferimento ad aree di memoria allocate al programma (non più riusabili)

```
int *p1,*p2;
     p1 = (int *) malloc(sizeof(int)); *p1 = 10;
     p2 = (int *) malloc(sizeof(int)); *p2 = 20;
     *p1 = *p2;    /*  SI:  *p1 == *p2 == 20  */
     p1 = p2;              /*  Da non fare!  */
```

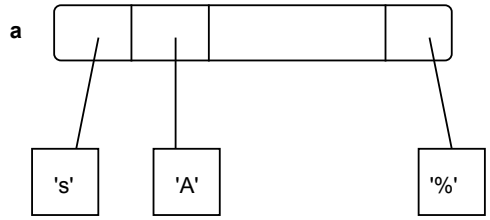
L'area puntata da p1 non è più raggiungibile, ma rimane allocata al programma!

NOTA PRECEDENZA *

[ ] HA PRECEDENZA RISPETTO A *

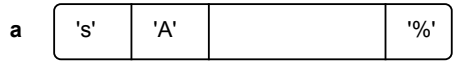
Quindi **char *a[ ];** ===> equivale a **char *(a[ ]);**

a è un *array di puntatori a caratteri*.



Per un *puntatore ad un array di caratteri*
è necessaria la parentesi

char (* a) [ ]



OPERAZIONI SU STRUTTURE DATI

Per i dati costruiti possiamo considerare le operazioni fondamentali di assegnamento e confronto (identità)

Tipo di dato strutturato	Assegnamenti	Confronti
array (costante)	no	no
strutture	si	no
unioni	si	no
puntatori a dati	si	si
puntatori a funzioni	si	si
puntatore a dati e puntatore a funzione (e viceversa)	no	no
puntatore a dati e array	si	no
array e puntatore a dati	no	no

Il **puntatore** è un costruttore **molto 'libero'**
pericolosamente vicino all'indirizzo

Vale il concetto di uguaglianza (dei puntati)

MANCANZE in DISCIPLINA

Le deroghe alla disciplina di *strutturazione* e *programmazione* in C sono introdotte

per la *programmazione* ***di sistema***

però possono portare a
programmazione *oscura* e di difficile *riusabilità*.

ancora di SISTEMA

Stretta relazione tra array e puntatori {sic}

Gli array ed i puntatori sono considerati come la stessa **notazione**
nome di un array ==
un puntatore al suo primo elemento,
➠ **possibilità di incremento/decremento sugli indirizz**i
dei puntatori

```
char v1[10],  *v2;
/* v1          è una costante e come nome equivale a &v[0]       */

/* v1 = v2; NO */
v2 = v1;

v1[0]                equivale a  *(v1)
v1[1]                equivale a  *(v1 + 1)
v1[expr]     equivale a  *(v1 + expr)
```


ARITMETICA SUGLI INDIRIZZI

Ogni riferimento ad un elemento di un array è espanso come un ***puntatore*** *dereferenziato* e spiazzamento rispetto al primo elemento

```
int  arr [10],    *puntarr;

puntarr = arr;
arr [0] equivale a *arr  ===>
arr [0] equivale a *puntarr

Non solo  * arr   ma /* risic */
* (arr + 2) si riferisce il terzo elemento
(anche se non è un primitivo)
data arrdata [5], *puntdate;
```

* **(arrdata + 1)** è la seconda data
nell'array arrdata costituito da 5 elementi data
equivale a **arrdata [1]**

il Compilatore **non controlla** ma fa solo eseguire la somma dell'indice (scalato) con l'indirizzo del primo elemento

a[i] e **i[a]** sono lo stesso
===> * (a + i) === * (i + a)

Esempio:

```
main ()
{ char a[]= "0123456789";
   int i = 5;
            printf ("%c %c %c %c\n", a[i], a[5], i[a], 5[a]);
}
```

si stampa:
5 5 5 5 {SIC**!!!**}
Ah, i risultati della equivalenza di nome

Questo non vuole dire che
array e puntatori siano equivalenti!

array

area di memoria allocata totalmente
(dimensioni fissate)
costante come nome

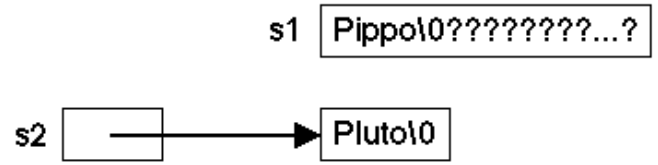
puntatori

area di memoria da allocare
(malloc o assegnamenti)
variabile con possibilità di cambiare valore

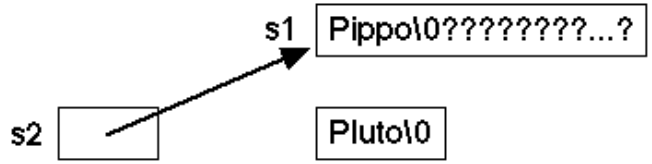
Esempio:

```
char s1[81] = "Pippo",*s2 = "Pluto";
```

s1 array di caratteri di **dimensioni fisse** (81) che può contenere una stringa di caratteri di **lunghezza variabile** (da 0 a 80 caratteri) - inizializzato a "Pippo"
s2 puntatore a carattere - inizializzato all'indirizzo del primo carattere della stringa "Pluto"



non posso scrivere s1 = s2; s1 è una costante
ma posso scrivere s2 = s1;
e se scrivo s2 = & s1; ottengo l'effetto precedente



L'area di memoria contenente la stringa "Pluto"
non è più referenziata e non è più referenziabile!

Esercizi

```
int *p1,p2[ ] = {1,2,3,4,5},k;
char *s1,s2[ ] = "12345",*s3 = "67890"

p1 = 0;   *p1 = 0;
p1 = (int *) 0;
p1 = 55;        p1 = (int *) 55;

printf("%d",p1);
printf("%d",*p1);
printf("%d",&p1);

s1 = "";
s1 = '\0';                  s1 = "\0";
s1 = "abc";    s1 = &"abc";

p1 = (int *) malloc(sizeof(int)*10);
s1 = (char *) malloc(81);

for (k = 0; k < 5; k++)  p1[k] = p1[k+5] = p2[k];

free(p2);

s3 = s1;
free(s3);
for (k = 0; k < 5; k++) *s1++ = s2[k];
```


Stringhe

DATO ASTRATTO (?)

RAPPRESENTAZIONE

- stringhe array di caratteri
- L'ultimo carattere di una stringa deve essere il carattere nullo (**\0** o **NULL**).

Esempi di stringhe

```
char s1[81];
char *s2;
```

s1 è una stringa di dimensione non fissata
s2 è un puntatore a carattere: deve essere fatta una allocazione esplicita
(non necessaria con la definizione precedente)

```
 s2 = (char *) malloc (81);
```

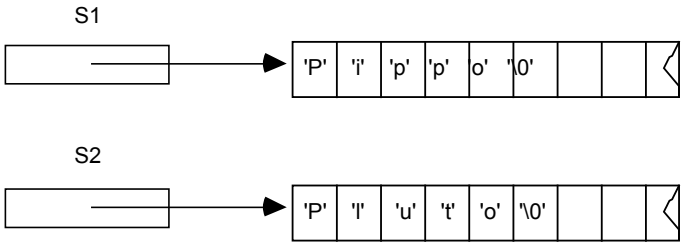
Potremmo **assegnare** da una all'altra, senza problemi

In realtà, il nome di un array è una costante

```
s1 = s2;   /* scorretta  */
s2 = s1;   /* corretta  */
```

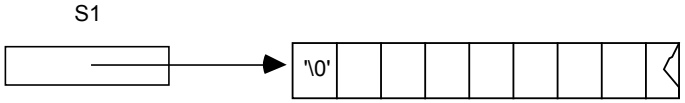
Stringhe

```
char s1[81];
char *s2;
```

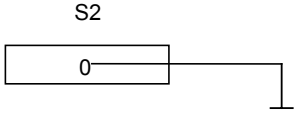


Distinguiamo tra memoria allocata
e lunghezza della stringa contenuta

Casi degni di nota



Stringa VUOTA

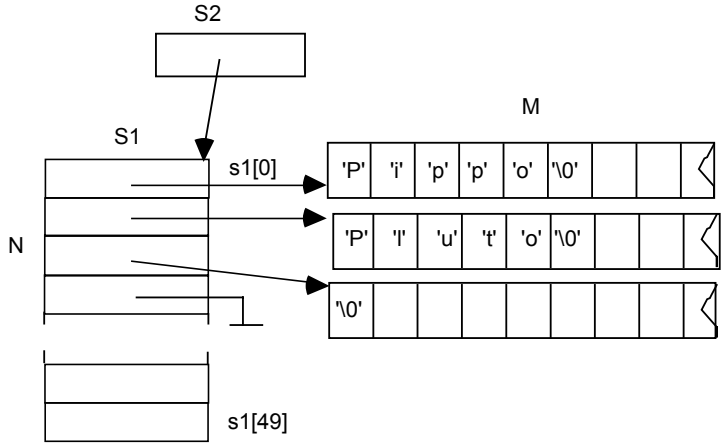


Stringa NULLA

Strutture di PUNTATORI

statiche e dinamiche

```
char ** s2;
char  * s1 [50];
```



Strutture statiche
Allocazione preparata dal compilatore
NxM locazioni contigue

Strutture dinamiche
Necessità di **allocazione** della memoria
Azioni esplicite

- di allocazione
- di aggancio ad aree esistenti

STRINGHE DI CARATTERI

Stringa == Array di caratteri

```
char  string[81];    /*  max 80 caratteri  */
```

L'ultimo carattere deve essere un NULL ('\0')

```
char  text[6] = {'P','l','u','t','o','\0'};
char  text[ ] = {'P','l','u','t','o','\0'};
char  text[ ] = "Pluto";
```

text	P	l	u	t	o	\0

Dove è allocata la memoria per le stringhe costanti?

Esempio - funzione **strlen** - libreria standard C
restituisce la lunghezza di una stringa di caratteri

```
strlen(char s[ ])
   {
   int j;
   for (j = 0; s[j] != '\0'; j++) ;    /*  ... ; s[j]; ...  */
   return j;
   }
```


Esempio - funzione **strcmp** - libreria standard C

confronta due stringhe di caratteri s1 e s2
restituisce un valore:

< 0	se s1 < s2
0	se s1 == s2
> 0	se s1 > s2

```
strcmp(unsigned char s1[ ],unsigned char s2[ ])
   {
   int j = 0;
   while (s1[j] && s2[j] && s1[j] == s2[j])  j++;
   return (s1[j] – s2[j]);
   }
```

Esempio - funzione **strcpy** - libreria standard C

copia una stringa (sorgente) in un'altra (destinazione)
restituisce l'indirizzo della stringa destinazione

```
char *strcpy(char dest[ ],char src[ ])
   {
   int j = 0;
   do
      dest[j] = src[j];
   while (src[j++])
   return dest;              /*  return &dest[0]  */
   }
```

PUNTATORI e ARRAY

STRETTA RELAZIONE TRA PUNTATORI E ARRAY
Array e puntatori sono (quasi) **equivalenti** come **nome**
nome di un array == puntatore al primo elemento

```
int a[10],*p;
```

ARITMETICA DEI PUNTATORI

Il compilatore C esegue sempre la conversione

a[k] ➠ *(a+k)

Si noti che, applicando l'operatore & ad entrambi i termini si ottiene:

&a[k] == a+k

a+k rappresenta la sequenza di interi che inizia al k-esimo posto

Il compilatore **non esegue alcun controllo** e genera sempre il codice per eseguire la somma dell'indirizzo del primo elemento dell'array con l'indice scalato

```
long int a[ ] = {100,200,300,400,500};
```

a[-1]	?	:A-4
a[0]	100	:A
a[1]	200	:A+4
a[2]	300	:A+8
a[3]	400	:A+12
a[4]	500	:A+16
a[5]	?	:A+20

UN PUNTATORE È UNA VARIABILE

```
p = a;      /*  *p == a[0]  */
p++;        /*  *p == a[1]  */
```

IL NOME DI UN ARRAY È UNA COSTANTE

```
a = p;      /*  NO!  */
a++;        /*  NO!  */
```

ARGOMENTO FORMALE SEMPRE VARIABILE

```
fun(..., int a[ ], ...) { ... }
fun(..., int *a, ...) { ... }        /*  Equivalenti  */
```

Esempio - funzione **strlen**
restituisce la lunghezza di una stringa di caratteri

```
strlen(char *s)
  {
  int j;
  for (j = 0; *s++; j++) ;
  return j;
  }
```

Esempio - funzione **strcmp**

confronta due stringhe di caratteri s1 e s2
restituisce un valore:

< 0	se s1 < s2
0	se s1 == s2
> 0	se s1 > s2

```
strcmp (unsigned char *s1,unsigned char *s2)
   {
   while (*s1 && *s2 && *s1 == *s2)  s1++, s2++;
   return (*s1 – *s2);
   }
```

Esempio - funzione **strcpy**

copia una stringa (sorgente) in un'altra (destinazione)
restituisce l'indirizzo della stringa destinazione

```
char * strcpy (char *dest,char *src)
   {
   char *p = dest;

   while (*dest++ = *src++)  ;
   return p;
   }
```


PUNTATORI vs. ARRAY

```
/*      array di puntatori a caratteri e puntatori di puntatori
        sono per alcuni aspetti equivalenti  */
#define MAX 100
#define NULL (char *) 0

void proc (n, arg)
/* procedura di stampa vettore o lista stringhe */
  int n; char ** arg;
{               int j;
        for (j=0; j<n; j=j+1)  /* array di stringhe */
        printf("Stringa %d vale %s\n", j, arg[j]);

        j=0;
        while (*arg)                    /* doppia lista  */
        {       printf("Stringa %d vale %s\n", j, *arg);
                j=j+1;          arg=arg+1;
}       }

main ()
{ int i,n;   char *s[MAX];
   printf("Dammi n\n");  scanf("%d", &n);
   for (i=0; i<n; i=i+1)
                {       s[i] = (char *) malloc (81);
            printf("Stringa %d ", i); scanf("%s", s[i]);
          }
   s[i] = NULL;   /* stringa nulla */
   proc(n, s);
}
```

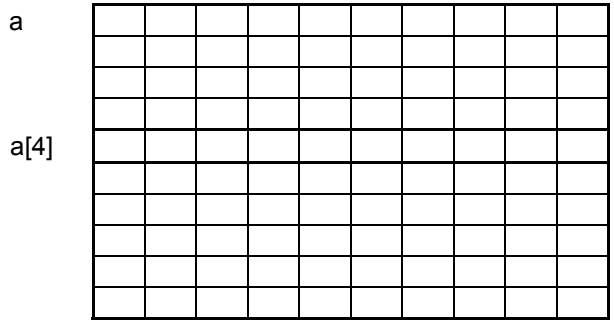
Esercizio:

```
int a[10][10],*b[10],(*c)[10];
```

qual è la differenza tra **a**, **b** e **c**?

a è una **matrice 10 x 10** di int

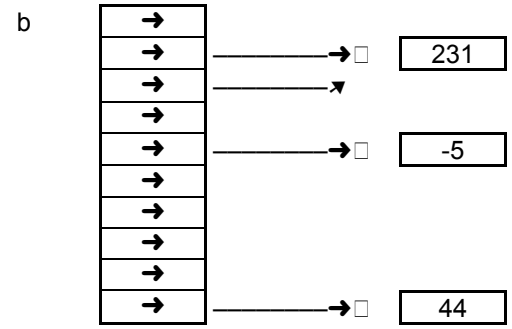
sizeof(a) è 200 == 10 x 10 x sizeof(int)
sizeof(a[k]) è 20 == 10 x sizeof(int)
sizeof(a[k][m]) è 2 == sizeof(int)



... a = ... /* NO! */
... a[k] = ... /* NO! */

b è un **array** di **10 puntatori** a int

sizeof(b) è 20 (o 40) == 10 x sizeof(int *)
sizeof(b[k]) è 2 (o 4) == sizeof(int *)
sizeof(*b[k]) è 2 == sizeof(int)

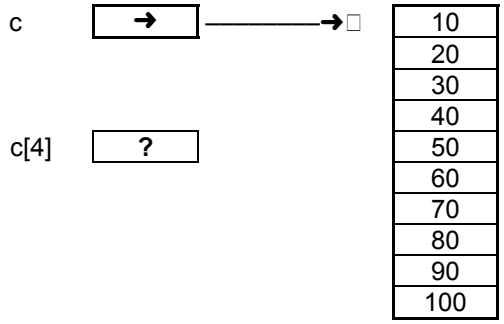


... b = ... /* NO! */
... b[k] = ... /* SI! */

```
b[1] = (int *) malloc(sizeof(int));  *b[1] = 231;
b[2] = b[1];   /*  Attenzione!  */
b[4] = (int *) malloc(sizeof(int));  *b[4] = -5;
b[9] = (int *) malloc(sizeof(int));  *b[9] = 44;
```

c è un **puntatore** a un **array di 10** int

sizeof(c) è 2 (o 4) == sizeof(int *)
sizeof(*c) è 20 == 10 x sizeof(int)
sizeof((*c)[k]) è 2 == sizeof(int)




```
...  c = ...            /*  SI!  */
...  c[k]  ...                /*  NO!  */
...  *c[k]  ...         /*  NO!  */
...  (*c)[k]  ... /*  SI!  */
```

```
c = (int *) malloc(10 * sizeof(int));    o meglio:
c = (int (*)[10]) malloc(sizeof(int [10]));
(*c)[0] = 10;
   for (k = 1; k <= 9; k++)  (*c)[k] = (*c)[k-1] + 10;
```


ATTENZIONE ALLE PRECEDENZE

Chiamata a procedura ()   Selezioni [ ] -> .
Unari! ~ + – ++ –– & * (type) sizeof

```
int *p;
```

qual è la differenza tra

```
*p++     *(p++)     (*p)++
```

Gli operatori ++ e * hanno la **stessa precedenza** e sono **associativi da destra a sinistra**

***p++** coincide con ***(p++)** e vuol dire:
usa il valore puntato da p, quindi incrementa p
(*p)++ vuol dire:
usa il valore puntato da p, quindi incrementa tale valore - p rimane immutato

```
void incrementa(int n,int *p)
  {     int k;
        for (k = 0; k < n; k++)  p[k]++;
  }

void incrementa(int n,int *p)   /*  Versione II  */
  {while (n– –  >  0)  (*p++)++;}
```

Variabili

VISIBILITÀ (MODULI E BLOCCHI)
possibilità di riferire la entità

TEMPO di VITA (BLOCCHI)
durata della entità all'interno del programma

i MODULI statici sono FILE
i BLOCCHI dinamici sono funzioni o blocchi

VISIBILITÀ

auto *automatiche (locali)*

locali ad un blocco (blocco o funzione)
La entità è visibile solo all'interno nel blocco
non visibile all'esterno
come **variabili locali ad una procedura**

extern *esterne (globali)*

dichiarazioni riferite a variabili globali
visibili a tutto il programma

static *statiche (globali)*

variabili statiche interne alle funzioni o moduli
sono visibili dove sono stati definite
ma non sono visibili all'esterno

default: **extern** per le variabili globali
auto per le variabili locali

MEMORIA STATICA e DINAMICA

Memoria **statica** ==>
variabili globali definite nel programma principale

Memoria **dinamica** ==>
variabili locali alle funzioni e con tempo di vita
pari alla esecuzione delle funzioni

Memoria **dinamica** ==>
variabili senza nome accedute attraverso puntatori

DATI STACK HEAP separati

Tramite funzioni di libreria (**malloc**)
analogamente deallocazione (**free**).

```
puntatore = (datatype*) malloc (sizeof (datatype));
```

La **malloc alloca** un certo **numero di byte,** fornendo **un puntatore** a questi
Il tipo è **puntatore a carattere**

```
int *ptr;
ptr = (int *) malloc (sizeof (int));
* ptr = 55;
```

Esempio automatiche e statiche

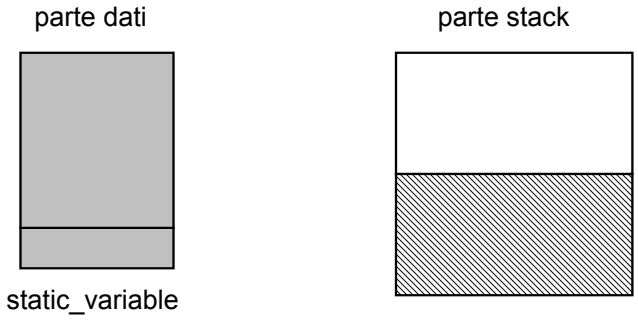
```
static_demo ();
main()
{  int i;
       for( i= 0; i < 10; ++i)            static_demo();
 }

static_demo()
{  int variable = 0;
   static int static_variable = 0;
printf("automatic = %d, static = %d\n",
                    ++variable,   ++static_variable);
}
```

variable automatica

visibile e **presente solo** durante la invocazione
sempre a 0 ad ogni invocazione

static_variable

viene allocata come globale (una volta sola) e
visibile solo durante la invocazione
qui è incrementata ad ogni chiamata



TEMPI DI VITA delle VARIABILI

Allocazione dei dati

- **automatici**: allocazione **locale**
  tempo di vita la procedura di definizione
  I dati sono locali al blocco di dichiarazione
  Allocazione e deallocazione al termine del blocco o procedura
  Politica realizzata tramite ***stack***

- **statici/extern**: allocazione **globale**
  tempo di vita pari al programma
  Una variabile statica interna ad una funzione permane oltre la singola invocazione della procedura.
  Ogni invocazione della stessa procedura utilizza il valore precedente della variabile.
  Politica di allocazione attraverso ***dati statici***

- **dinamici**: allocazione **dinamica** dei dati riferiti attraverso puntatori
  tempo di vita dipendente dall'utente ma non legato ad un puntatore specifico, ma ad azioni di deallocazione
  l'area di memoria deve essere esplicitamente allocata/deallocata, usando le funzioni del sistema operativo (**malloc/free**)
  Politica realizzata attraverso una gestione di **memoria** ad ***heap***

Classi di memorizzazione

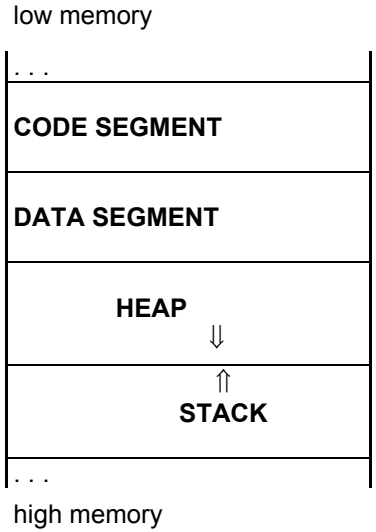
TEMPI di VITA
VISIBILITÀ

Ogni entità (variabile o funzione) ha:

- un **NOME** che la identifica (in modo univoco ?)
- un **TIPO** che identifica l'insieme dei valori ammessi e la rappresentazione interna della variabile o del risultato della funzione
- un **VALORE** tra quelli ammessi dal tipo
- un **INDIRIZZO** relativo al primo byte del blocco di memoria che contiene il valore della variabile o il codice della funzione
- una **CLASSE di MEMORIZZAZIONE** che indica il tipo di area di memoria in cui la variabile o la funzione viene memorizzata

dati	funzioni
DATA SEGMENT STACK HEAP REGISTRI	CODE SEGMENT

Un esempio di Struttura della memoria a RUN-TIME



Come scoprire eventuali collisioni tra STACK e HEAP?

- il S.O. del Macintosh chiama 60 volte al secondo lo "stack sniffer"
- il Turbo C++ ha un opzione in compilazione "Test Stack Overflow"

CLASSE di MEMORIZZAZIONE auto

- automatica - **default** per **variabili locali**, non si applica alle funzioni
- **visibilità locale**: la variabile è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
- la variabile è **temporanea**: esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- su **STACK (**valore iniziale di default ?)

```
somma(int v[ ],int n)
{
auto int k,sum = 0;    /*  Quanto vale k ?  */
for (k = 0; k < n; k++) sum += v[k];
return sum;
}

fattoriale(int n)  /*  solo n >= 0  */
{
if (n <= 1) return 1;
else return n * fattoriale(n - 1);
}
```

... fattoriale(4) ...

	...	4	...	3	...	2	...	1	

CLASSE di MEMORIZZAZIONE register

- come le auto
- su **REGISTRO MACCHINA**

```
somma(int v[ ],register int n)
{
register int k,sum = 0;
for (k = 0; k < n; k++) sum += v[k];
return sum;
}


fattoriale(register int n)     /*  solo n >= 0  */
{
if (n <= 1) return 1;
else return n * fattoriale(n - 1);
}
```

Cosa guadagno in quest'ultimo caso?

CLASSE di MEMORIZZAZIONE extern

- esterna - **default** per **variabili globali** e **funzioni**
- **visibilità globale**: visibile ovunque, dal punto di definizione (o dichiarazione) in poi
  visibile anche **al di fuori del file** che ne contiene la definizione
- **permanente**: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- se dati, inizializzati a 0
- su **CODE SEGMENT** (**funzioni**) oppure
- su **DATA SEGMENT** (**variabili** - valore iniziale di default 0)

File "AAA.c"

```
extern void fun2(...);
...
int ncall = 0;
...
fun1(...)
{
ncall++;
...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...
void fun2(...)
  {
  ncall++;
  ...
  }
```

la variabile ncall e le funzioni fun1 e fun2 sono visibili ed utilizzabili in entrambi i file

CLASSE di MEMORIZZAZIONE static

statica - definizione globale o locale

visibilità:

- **globale** nel caso di definizione globale: visibile ovunque, dal punto di definizione (o dichiarazione) in poi, ma **solo all'interno del file che la contiene**
- **locale** nel caso di definizione locale (solo variabili): visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi

permanente: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
su **DATA SEGMENT** (**variabili** - valore iniziale di default 0) oppure
su **CODE SEGMENT** (**funzioni**)
se dati, inizializzati a 0

File "CCC.c"

```
fun1(...);
funA(void);
extern funB(void);
static int ncall = 0;
...
static fun1(...)
  { ncall++; ... }
funA(void)
  { return ncall; }
```

File "DDD.c"

```
void fun1(...);
funB(void);
extern funA(void);
static int ncall = 0;
...
static void fun1(...)
  { ncall++; ... }
funB(void)
  { return ncall; }
```

CODE SEGMENT

- le funzioni nel segmento codice

DATA SEGMENT

- variabili extern (globali multi-file)
- variabili static (globali single-file e locali)

STACK

- variabili auto (locali - argomenti funzioni)

REGISTRI

- variabili register (locali - argomenti funzioni)
  --- non tutti i tipi di variabili ---

HEAP

- strutture dati allocate (malloc) e deallocate (free) esplicitamente dall'utente e referenziate tramite puntatori

APPLICAZIONE SU PIÙ FILE

Compilazioni indipendenti dei file + **collegamento**

Durante la compilazione di un file sorgente, il compilatore **non vede le entità** (variabili e funzioni) **definite negli altri file** ➠ è necessario **dichiarare le entità esterne** utilizzate

DICHIARAZIONE: fornisce l'**interfaccia** a una
funzione (prototipo), **dato** o **tipo di dato**
--- non viene allocato spazio in memoria

```
extern fattoriale(int n);
extern float xyz(...);/*  in altro modulo  */
extern int ncall;      /*  in altro modulo  */
typedef short int Signed16;
```

DEFINIZIONE: fornisce l'**implementazione** di una
funzione o **dato**
--- viene allocato spazio in memoria

```
fattoriale(int n) {...}
int ncall = 0;          /*  globale  */
```

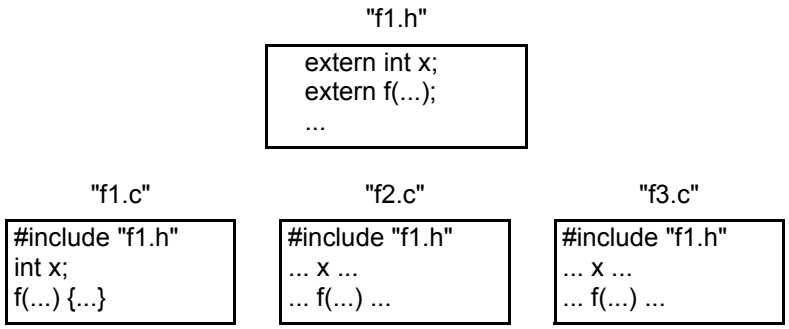
Una DEFINIZIONE può fungere anche da DICHIARAZIONE

Ogni entità può essere dichiarata *più volte* (in file diversi) ma deve essere definita *una e una sola* volta

Il file "f1.c" **mette a disposizione** la variabile x e la funzione f() - DEFINIZIONI

I file "f2.c" e "f3.c" **utilizzano** la variabile x e la funzione f() messa a disposizione dal file "f1.c" - DICHIARAZIONI

Tutte le **dichiarazioni** possono essere inserite **in un HEADER FILE** "f1.h" incluso dai file utilizzatori



Un **header file** contiene SOLO **dichiarazioni** e NON **definizioni**

POSSIBILITÀ DI SVILUPPARE UN PROGRAMMA SU PIÙ FILE: VISIBILITÀ E PROTEZIONE

VARIABILI GLOBALI

static

==> entità allocate una volta per tutte, all'inizio del programma, ***visibili solo nel file di definizione***.

Una variabile/funzione statica definita a livello di file non è visibile al di fuori del file stesso.

Ruolo simile a quello di una **variabile/funzione *protetta e non visibile* di un modulo**

extern

==> entità dichiarate all'interno di un file per riferire entità definite in altri file
o
esportate perché possano essere riferite da altri

La clausola extern quindi è usata
sia da chi la importa
sia da chi le esporta, seppure con semantica diversa

La classe **extern** è il **default** per ogni entità dichiarata/ definita a livello di programma.

Le dichiarazioni di extern sono simili alle variabili/funzioni **importate** da un modulo/unità
Le definizioni di entità extern corrispondono agli **export**.

chi esporta la entità, la definisce
chi importa la entità, la dichiara

Le dichiarazioni in altri file servono per collegarsi alle stesse variabili/funzioni e consentire controlli al compilatore

METODOLOGIA DI USO

le *definizioni* non usano extern ed usano il default: non compare la clausola esplicitamente

le *dichiarazioni* riportano la classe *extern*

Si noti che l'utente non conosce i file di importazione

Esempio: Uno stack

Uno stack che contiene valori reali ===>
data abstraction con funzioni push, pop e isempty, isfull.

La data abstraction stack, in questo caso, è racchiusa tutta in un solo file (*stack.c*) che viene usato dai programmi che ne abbiano necessità
===> **CONDIVISIONE DI DEFINIZIONI**

```
#define DIM       40
#define true      1
#define false     0

static int top = 0;                /* inizializzazione */
static double stack [DIM];
/* strutture  dati non visibili al  di fuori del  file stack.c */
/* le funzioni seguenti sono assunte external a default  */
int isempty () { return ((top)? false: true); }
int isfull  () { return ((DIM - top)? false: true); }

void push ( f)      /* definizione della funzione esportata */
      double f;     /*  PROTOTIPO:  void push (double f) */
{  stack [top++] = f;   /* non si tratta l'overflow */  }

double pop () {
      double f;
       f = stack [- -top];  /* non si tratta underflow */
       return f;       }
```

Il programma principale, **specificato in un file diverso**, deve dichiarare **extern** le funzioni da **importare**, ma **non può accedere** direttamente alla rappresentazione dello stack (dichiarata **static**)

Ad esempio, nel file *progr.c*:

```
extern int isempty (), isfull ();       /* dichiarazioni per l'uso */
extern void push ();                    /* locale delle funzioni   */
extern double pop ();
main () { double a, b; int c;
      do {    printf ("vuoi fare la push: si 1/ no 0\n");
              scanf  ("%d", &c);
              if ((! isfull ()) && (c != 0))
                    {       printf (" valore da inserire\n");
                            scanf  ("%le", &a);   push (a); }
              else if (c!=0) puts("stack pieno");
```

```
                printf (" vuoi fare la pop: si 1/ no 0\n");
        scanf  ("%d", &c);
        if ((! isempty ()) && (c != 0))
                        {       b = pop ();
                                printf ("\n valore estratto   %e",b); }
        else if (c!=0) puts ("stack vuoto");
                printf (" vuoi uscire: si 1/ no 0\n");
        scanf  ("%d", &c);
    }
  while (c != 1);
}
```

Per ottenere un **unico eseguibile** ===>
bisogna fare il **LINKING** dei file ***stack.c*** e ***progr.c***

Se volessimo mettere il tutto (cioé data abstraction stack e programma main) su **un solo file**, e garantire la **stessa protezione** dovremmo operare in questo modo:

```
#define DIM 40
#define true 1
#define false 0
extern int isempty(), isfull();
extern void push();  /* dichiarazioni per il main */
extern double pop();

main () {
double a, b; int c;
        do {    ...
             }    /* come sopra */
while (c != 1);
}

static int top = 0;                 /* inizializzazione */
static double stack [DIM] ;

/*  le  dichiarazioni statiche sono visibili nell'ambito del file di definizione
      solo dopo la definizione stessa (e non prima). Il main quindi non
      può accedervi  direttamente
 */
int isempty () { return ((top) ? false: true); }
int isfull  () { return ((DIM - top)? false: true); }
void push (double f)  { stack [top ++] = f; }
       double pop () { double f; f = stack [top- -];  return f; }
```

Ancora l'esempio dello STACK
utilizzando il concetto di file **HEADER**:

File *stack.h*: **INTERFACCIA DELLO STACK**

```
extern int isempty(), isfull();
extern void push();      /* dichiar. da condividere */
extern double pop();
```

File *stack.c:*

```
#define DIM 40
#define true 1
#define false 0

static int top = 0;                /* inizializzazione */
static double stack [DIM] ;

int isempty () { return ((top) ? false: true); }
int isfull  () { return ((DIM - top)? false: true); }
void push (double f)  { stack [top ++] = f; }
double pop ()
        { double f;   f = stack [top- -];  return f; }
```

File *progr.c:*

```
#include "stack.h"
main () { double a, b; int c;
            do {    ...          }    /* come sopra */
while (c != 1)}
```

Per produrre un **unico eseguibile** ===>
bisogna **LINKARE** progr.c e stack.c **assieme**

Esempio: Una lista

Nel seguito si codifica l'esempio della funzioni relative ad una lista questa volta specificate su **più file**.

Primo file: INTERFACCIA di una LISTA
file ***list.h*** che i programmi per usare le liste devono includere cioé **IMPORTARE**

```
/*  tutte le funzioni seguenti sono dichiarate esterne */
extern void Create (), End (), Enqueue(int i);
extern void EnqueueF(int i), EnqueueL(int i);
extern int  DequeueF (), DequeueL (), Dequeue (int i);
extern int  IsIn (int i), Empty (), Length ();
```

Secondo file: IMPLEMENTAZIONE di una LISTA
file ***list.c*** che contiene l'implementazione di tutte le funzioni di lista

```
#include <stdio.h>
#include <alloc.h>
#define NULL 0

struct node              /* elemento della lista */
     {int                item;
      struct node    *next;
     };
static struct node *first, *last;
/*puntatori agli elementi iniziale e finale*/
```

```
void Create ()  /* inizializza la coda */
{     first = NULL;  last = NULL;   }

void End ()     /* riazzera la coda  */
{     int    i;
      while (first != NULL) i = DequeueF();
}

int  IsIn (int i)
/* verifica la presenza di un elemento nella coda */
{   struct node *t;
      t = first;
      while (t != NULL && t ->item != i) t = t ->next;
      return(t != NULL);
}

int  Empty ()   /* verifica se la coda è vuota o meno */
{     return (first == NULL);  }

int  Length ()  /* numero degli elementi in lista */
{   int count = 0;
      struct  node *temp = first;
      while (temp != NULL) { count++; temp = temp -> next;}  return
      (count);
}
```



```
void EnqueueF (int i)  /* aggiungi al primo posto in coda */
{   struct node *newnode;
newnode = (struct node *) malloc(sizeof(struct node));
newnode ->next   = first;              newnode ->item   = i;
if (first == NULL) { last = newnode;}
first            = newnode;
}

void EnqueueL (int i)
/* aggiunta all'ultimo posto in coda */
{   struct node *newnode;

      newnode = (struct node *) malloc(sizeof(struct node));
      newnode ->next    = NULL;
      newnode ->item   = i;
      if (first == NULL)
            {   first          = newnode;
              last                  = newnode; }
      else
       {    last->next = newnode;
              last           = newnode; }
}

void Enqueue (int i)
/* aggiunta solo se non c'e' in coda */
       {   struct node *t = first;
       while (t != NULL && t -> item != i) t = t -> next;
       if (t == NULL) /* inserisci l'elemento */
              EnqueueF (i);
}
```

```
static Dealloca (struct node *temp)
/* funzione PRIVATA non visibile all'esterno del file*/
{ int reply;
        reply = temp -> item;
        free((char *) temp);
        return reply;
}

int DequeueF  ()    /* togli il primo elemento in coda */
{   struct node *temp = first;
      if (first == NULL) /* coda vuota */ return (NULL);
      else {     if (first == last)
/* un solo elemento da togliere */
                        {     first  = NULL;  last   = NULL; }
                 else   first = temp -> next;
                 return Dealloca(temp);  }
}

int DequeueL ()   /* togli l'ultimo elemento in coda   */
{   struct node *old, *new, *temp = last;
      if (first == NULL)  /* lista vuota */ return (NULL);
      else { if (first==last)
                 /* è il primo elemento da togliere */
                 {   last  = NULL; first = NULL; }
          else  /* si lascia almeno un elemento */
                 { old = first; new = old -> next;
                    while (new != last)
                        { old = new; new = new -> next;}
               last = old;  old ->next = NULL; }
      return Dealloca(temp);
}
```



```
int Dequeue (int i)
{       if (first == NULL)  /* lista vuota */ return (NULL);
       else { if (first -> item == i)
                        /* è il primo elemento da togliere*/
                              return Dequeuef();
                 else {       struct node *t, *temp;
                                    t = first; temp = t -> next;
                       while (temp !=NULL && temp ->item != i)
                                 { t = temp; temp = t -> next}
                              if (temp != NULL)  /*l'elemento c'e' */
                                          {       if (t -> next == last) last = t;
                                          t -> next = temp -> next;
                                          return Dealloca(temp);
                                                }
                              else return NULL;
                       /* l'elemento non c'e' */
                       }
}
```

ESEMPIO di PROGRAMMA che usa la LISTA:

file *prova.c*:

```
#include "list.h"
#include <stdio.h>

main ()
{ int .....;
      printf
      ("inizio programma di prova della lista su piu' file\n");
      Create();
      EnqueueF(12);
      .....
      DequeueL();
      .....
      End();
}
```

Anche in questo caso per creare un **UNICO ESEGUIBILE** ===>
bisogna **COLLEGARE** i file ***prova.c*** e ***list.c*** insieme