

Il puntatore

E' un tipo scalare, che consente di rappresentare gli **indirizzi** delle variabili allocate in memoria.

Il dominio di una variabile di tipo puntatore è un insieme di indirizzi:

☞ Il valore di una variabile di tipo puntatore puo` essere l'indirizzo di un'altra variabile (variabile *puntata*).

In C i puntatori si definiscono mediante il costruttore *.

Definizione di una variabile puntatore:

```
<TipoElementoPuntato> *<NomePuntatore>;
```

dove:

- <TipoElementoPuntato> e` il tipo della variabile puntata
- <NomePuntatore> e` il nome della variabile di tipo puntatore
- il simbolo * e` il costruttore del tipo puntatore.

Ad esempio:

```
int *P; /* P è un puntatore a intero */
```

Il puntatore

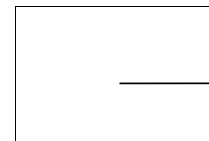
Operatori:

- Assegnamento: e' possibile l'assegnamento tra puntatori (dello stesso tipo). E' disponibile la costante NULL, per indicare l'indirizzo nullo.
- operatore di *dereferencing* *: è un operatore unario. Si applica a un puntatore e restituisce il valore contenuto nella cella puntata => serve per accedere alla variabile puntata.
- Operatore **Indirizzo** & si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale e' allocata la variabile.
- operatori *aritmetici* (vedi *vettori & puntatori*).
- Operatori relazionali: >, <, ==, !=

Ad esempio:

```
int *punt1, *punt2;  
int A;  
punt1=&A;  
*punt1=127;  
punt2=punt1;  
punt1=NULL;
```

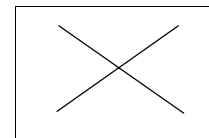
punt2



A



punt1



Un Esempio

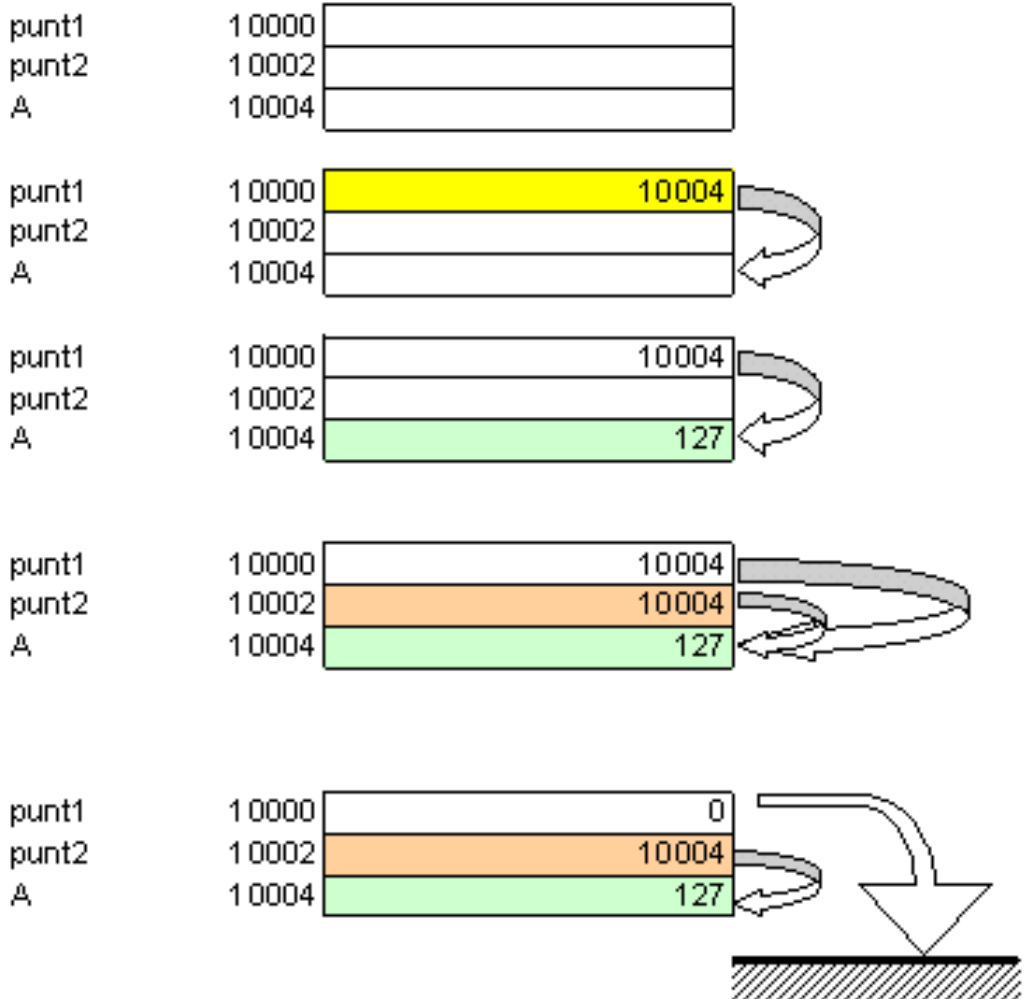
```
int *punt1, *punt2;  
int A;
```

```
punt1=&A;
```

```
*punt1=127;
```

```
punt2=punt1;
```

```
punt1=NULL;
```



Operatore Indirizzo &:

☞ & si applica solo ad *oggetti che esistono in memoria* (quindi, già definiti).

☞ & non è applicabile ad espressioni.

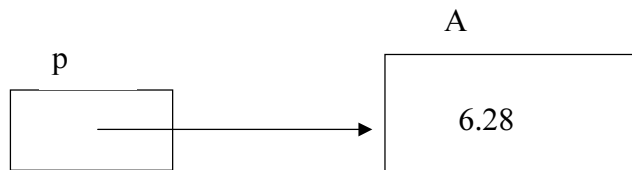
Operatore Dereferencing *:

☞ consente di accedere ad una variabile specificando il suo indirizzo

☞ l'indirizzo rappresenta un modo alternativo (alias) al nome per accedere e manipolare la variabile:

```
float *p;  
float R, A;
```

```
p=&A; /* *p è un alias di A*/  
R=2;  
*p=3.14*R; /* A è modificato */
```



Puntatore come costruttore di tipo

Dichiarazione di un tipo puntatore:

```
typedef <TipoElementoPuntato> *<NomeTipo>;
```

- <TipoElementoPuntato> è il tipo della variabile puntata
- <NomePuntatore> è il nome del tipo dichiarato.

Ad esempio:

```
typedef float *tpf;  
tpf p;  
float f;  
p=&f;  
...
```

Puntatori

Nella definizione di un puntatore e' necessario indicare il tipo della variabile puntata.

➔ il compilatore puo' effettuare controlli statici sull'uso dei puntatori.

Esempio:

```
typedef struct{...}record;
int *p, A;
record *q, X;

p=&A;
q=p; /*warning!*/
q=&X;
*p=*q; /* errore! */
```

☞ Viene segnalato dal compilatore (*warning*) il tentativo di utilizzo congiunto di puntatori a tipi differenti.

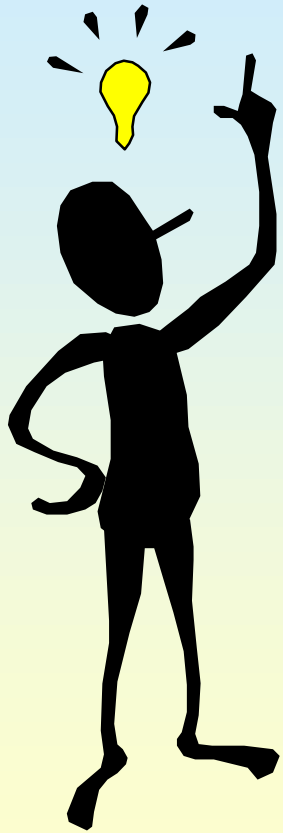
Con i puntatori possiamo considerare tre possibili valori:

pointer contenuto o valore della variabile pointer
(indirizzo della locazione di memoria a cui punta)

&pointer indirizzo fisico della locazione di memoria del puntatore

***pointer** contenuto della locazione di memoria a cui punta

Considerazioni sui Puntatori



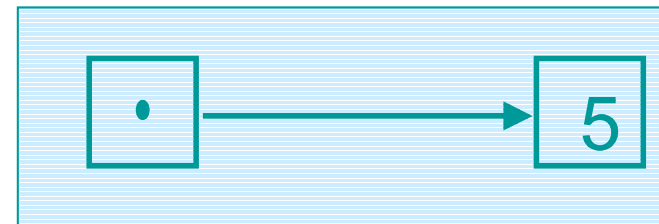
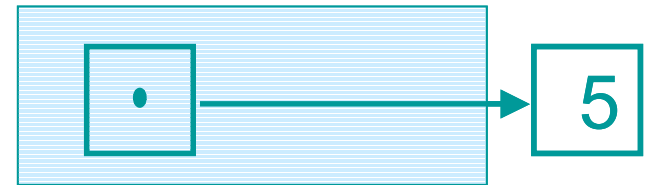
`int *iptr;`

`iptr` di tipo `int *`

`*iptr` di tipo `int`

`int *` `iptr;`

`int` `*iptr;`



EFFETTI COLLATERALI CON L'USO DI VARIABILI PUNTATORE: modifica non voluta di valori di variabili puntate

```
int *P;  
int    x, y;  
  
P=&y;  
x=3;  
*P=x;  
/* y vale 3*/
```

```
int *P, *Q;  
int    x, y;  
  
P=&x;  
Q=&y;  
*P=3;    /*x vale 3*/  
*Q=5;    /*y vale 5 */  
  
P=Q;     /*P punta a y*/  
*Q =7;   /* anche *P vale 7 */
```


Pointer Arithmetic

- **A pointer is a variable, so we can manipulate it**
- **but, number of operations is limited**

increment, decrement

```
++ptr; --ptr; ptr++; ptr--;
```

add, subtract integer amount

```
ptr += 5; ptr -= 2;
```

subtract one pointer from another

```
offset = ptr2 - ptr1; /* è un intero!!!! */
```

compare pointers

```
if( ptr1 != ptr2 )
```

```
if( ptr1 > ptr2 )
```

Puntatori a strutture:

E' possibile utilizzare i puntatori per accedere a variabili di tipo struct.

Ad esempio:

```
typedef struct{ int Campo_1,Campo_2;
                } TipoDato;

TipoDato  S, *P;

P = &S;
```

Il punto della notazione postfissa ha **precedenza** sull'operatore di dereferencing *; per accedere alle componenti della struttura referenziata da P è necessario utilizzare le parentesi tonde:

```
(*P).Campo_1=75;
```

Operatore ->:

L'operatore -> consente di accedere ad un campo di una struttura referenziata da un puntatore in modo più sintetico:

```
P->Campo_1=75;
```

```
typedef struct {  
    char        cognome[30];  
    char        nome[30];  
    data        data_di_nascita;  
    char        codice_fiscale;  
} dati_anagrafici;
```

```
dati_anagrafici    *P, S;  P=&S;
```

Accesso ai campi della struttura: le notazioni sono equivalenti

```
(*P).data_di_nascita.giorno           Precedenze  
(*P).cognome[0]
```

```
P - > data_di_nascita.giorno
```

```
P - > cognome[0]    /* senza usare il nome S */
```

Precedence

- Use parentheses when in doubt or to improve readability:

Level	Operator
16L	-> • [] ()
15R	sizeof ++ -- ~ ! + - (cast) * indiretto &indirizzo
13L	* / %
12L	+ -
11L	<< >>
10L	< <= > >=
9L	== !=
8L	& and bitwise
7L	^ xor bitwise
6L	or bitwise
5L	&& AND logico
4L	OR logico
2R	= *= /= %= += -= <<= >>= &= = ^=
1L	, virgola

Irwin Sheer

Superconducting Super Collider Laboratory

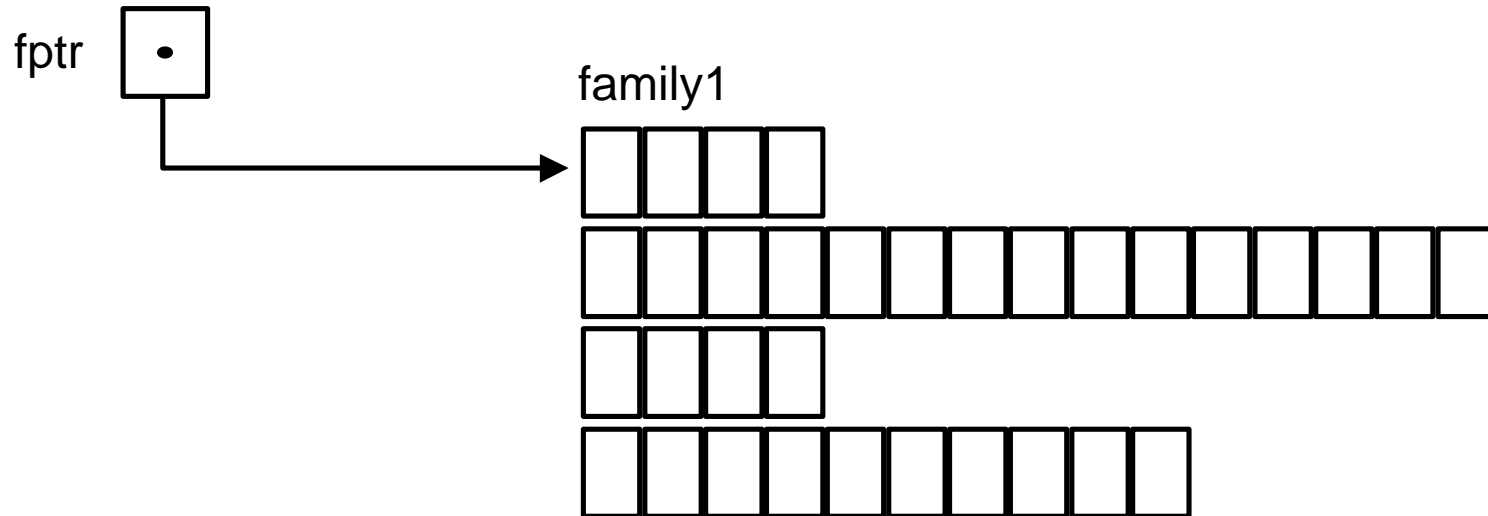
MS 2300, 2550 Beckleymeade Ave., Dallas, TX 75237

Tel: (214) 708-1050; Fax: (214) 708-6354

e-mail: Irwin_Sheer@ssc.gov

Pointer to a Struct

```
Given: struct family_rec {  
        int id_num;  
        char name[15];  
        int num_members;  
        double income;  
    } family1, *fptr;  
  
    fptr = &family1;
```



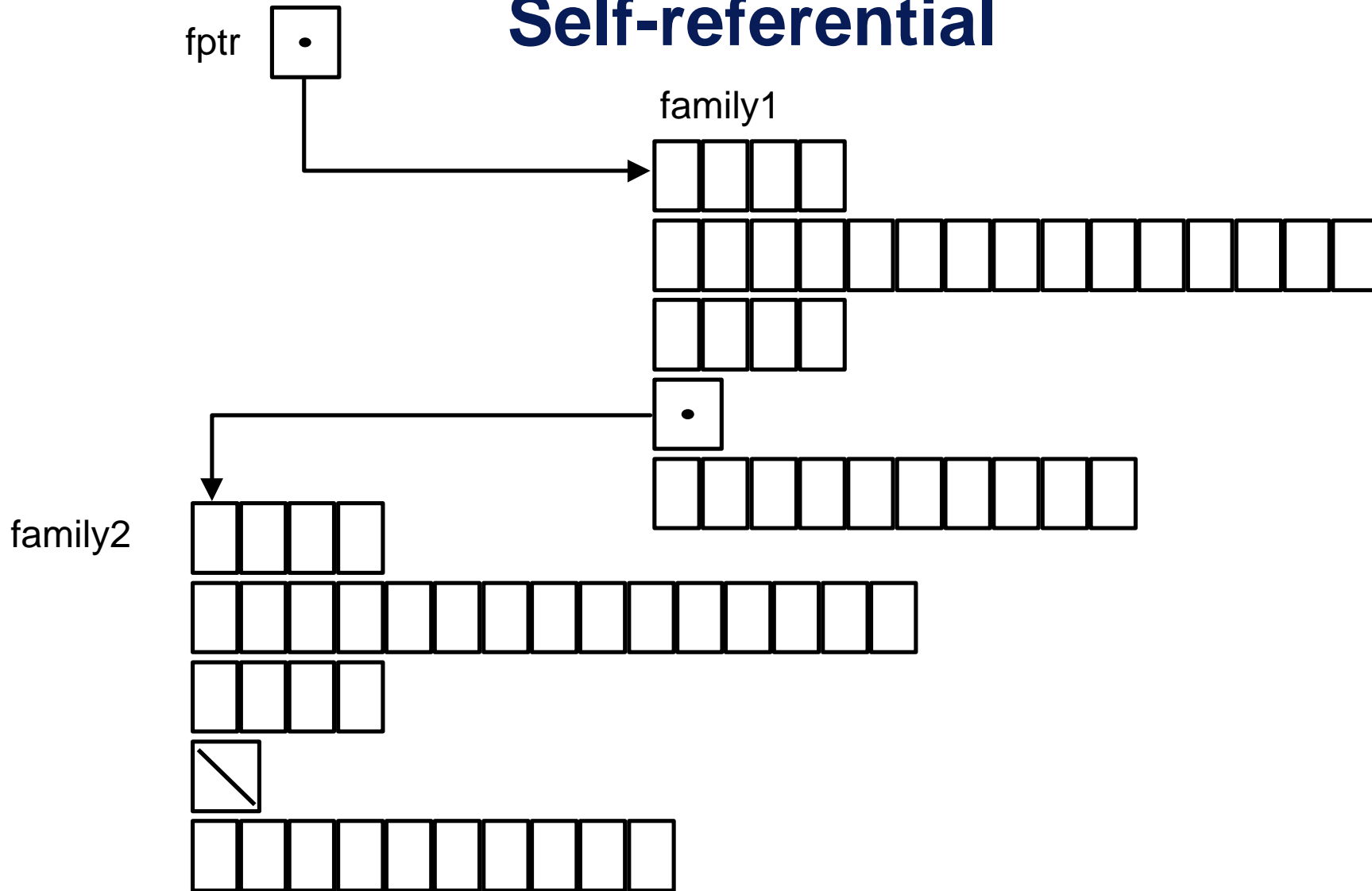
Self-referential Structs

Given **struct family_rec** {
 int id_num;
 char name[15];
 int num_members;
 ⇒ **struct family_rec** * next;
 double income;
} family1, family2, *fptr;

```
fptr = &family1;  
family1.next = &family2;  
family2.next = NULL;
```


Structs in C

Self-referential



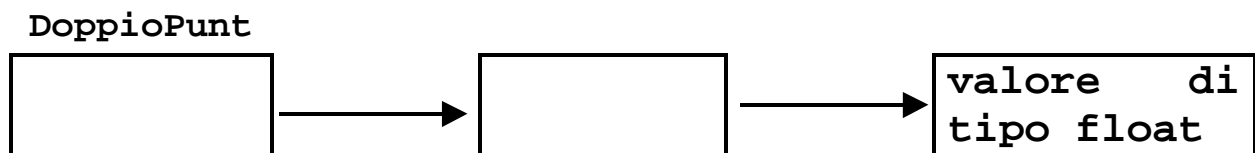
DOPPIO PUNTATORE

```
float    **DoppioPunt;
float    *Punt;
float    v;
```

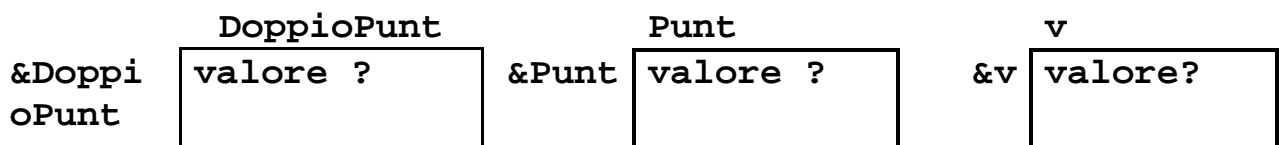
Significato:

P è una variabile puntatore che contiene l'indirizzo di un'area di memoria che, a sua volta, contiene l'indirizzo di un tipo float.
(**P** punta a un puntatore a float)

Rappresentazione

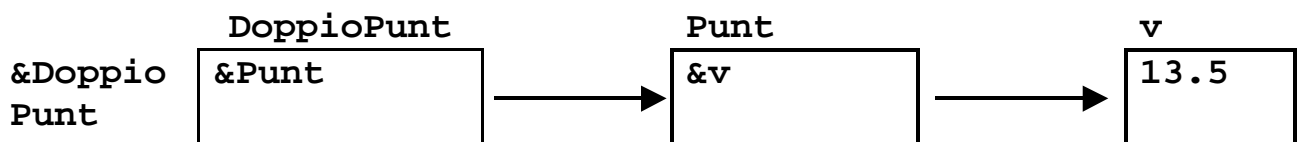


Dopo la dichiarazione:



```
DoppioPunt = &Punt;
*DoppioPunt=&v;
**DoppioPunt = 13.5;
```

Dopo gli assegnamenti



Levels of Indirection

- “**pointer to int**” holds address of int

```
int number = 5, new_number;
```

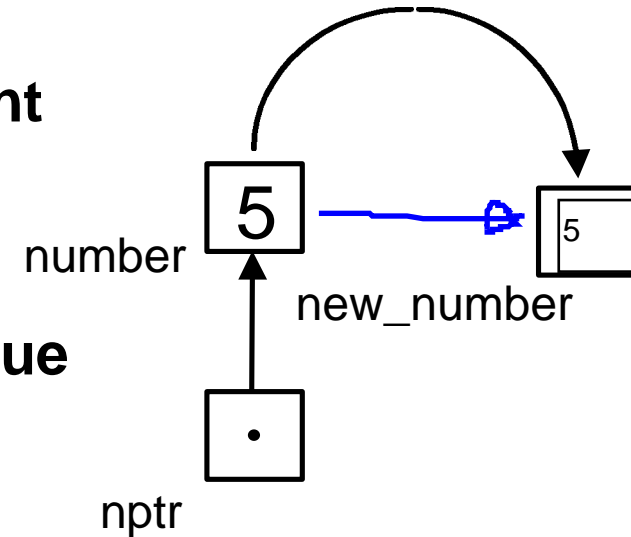
```
int *nptr = &number;
```

- **dereference pointer to “reach” value indirectly**

```
new_number = *nptr;
```

- **dereferencing a pointer to access the value is called indirection**
- **each dereference operator (*) that is applied gives 1 level of indirection**

*nptr uses only one level; all we need to get to the data value



Levels of Indirection

- can have multiple levels of indirection

```
int number = 5, new_number;
```

```
int *nptr = &number;
```

```
int **n2ptr = &nptr;
```

- **dereference same number of times as in declaration to reach value**

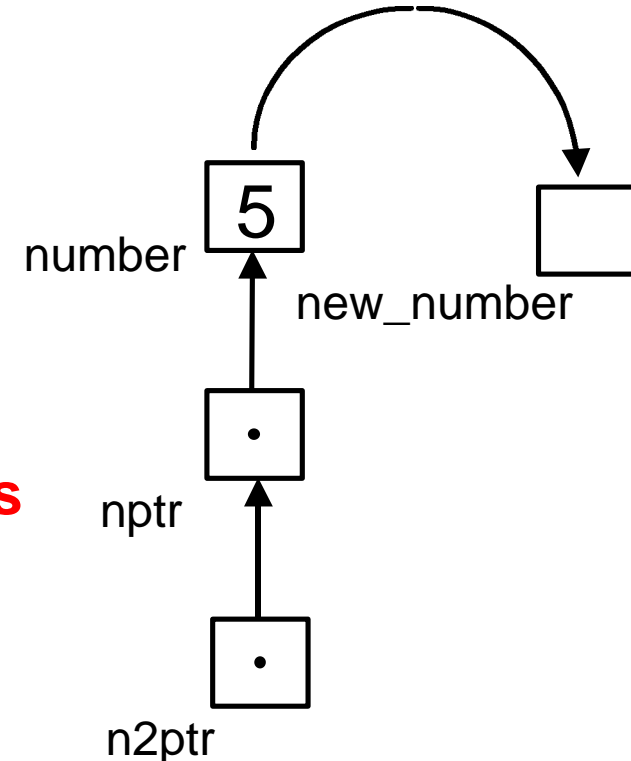
```
new_number = **n2ptr;
```

- each * is read as “pointer to”

number data type is int

nptr data type is “pointer to” int

n2ptr data type is “pointer to pointer to” int



Vettori & Puntatori

Vettori:

- in C, i vettori vengono allocati in memoria in **parole consecutive** (cioè parole fisicamente adiacenti), la cui *dimensione* dipende dal tipo degli elementi del vettore.
- Il *nome* di una variabile di tipo vettore viene considerato dal C come *l'indirizzo* del primo elemento del vettore.

Ad esempio:

```
int V[10];
```

☞ **V è una costante:**

- V equivale a **&V[0]**
- come tipo è un puntatore ad intero:

```
int *p, V[10];
p=V; /* p punta a V[0] */
V = p; /*NO! V è un puntatore costante*/
```

Vettori & Puntatori

Il C consente di eseguire operazioni di somma e sottrazione sui puntatori (a vettori).

Operatori aritmetici su puntatori a vettori:

Se V e W sono puntatori ad elementi di vettori ed i è un intero:

- (V+i) restituisce l'indirizzo dell'elemento spostato di i posizioni in avanti rispetto a quello indicato da i;
- (V-W): restituisce l'intero che rappresenta il numero di elementi compresi tra V e W.

Ad esempio:

```
float V[100], *p, *q;
int k;
p=V+7; /* p punta a V[7] */
q=V+2; /* q punta a V[2] */
k=p-q; /* k vale 5 */
...
```

VARIABILI PUNTATORE E ARRAY: ARITMETICA DEGLI INDIRIZZI

```
int i;
int vett[10];
int *P;
```

vett:

- è l'indirizzo del primo byte dell'array di interi (indirizzo primo byte di un intero)
- ha associato il tipo degli elementi
- ha un valore costante

vett si comporta come un puntatore «fisso»

vett[i]	equivale a	*(vett+i)
*(P+i)	equivale a	P[i]
P=vett	equivale a	P=&vett[0]
P=vett+i	equivale a	P=&vett[i]

(P+i) - (P+j) valore intero pari al numero di elementi (interi) tra i e j

Use with Arrays

- Closely related
- **Array name is address constant**; like a “pointer constant”
- **Array syntax vs. pointer syntax**
`char publisher[20];`
`publisher[12]` equivalent to `*(publisher + 12)`
- **Can also index with pointer**
`char *cptr = publisher /* costante di tipo pointer senza & */;`
`publisher[12]` same as `cptr[12]`

Vettori e Puntatori

- In C, ogni riferimento ad un elemento di un vettore è espanso come un *puntatore dereferenziato*:

V[0]	equivale a	*(V)
V[1]	equivale a	*(V + 1)
V[i]	equivale a	*(V+i)
V[expr]	equivale a	*(V + expr)

Ad esempio:

```
main ()
{
  char a[] = "0123456789"; /*a è un
                             vettore di
                             caratteri */

  int i = 5;

  printf("%c%c%c%c\n", a[i], a[5], i[a], 5[a]);
}
```

Stampa:

5 5 5 5

- Per il compilatore V[i] e i[V] sono lo stesso elemento, perché viene sempre eseguita la conversione:

V[i] => *(V+i)

senza eseguire alcun controllo ne' su V ne' su i.

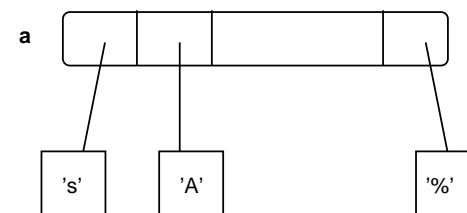
Vettori & Puntatori

- [] ha precedenza rispetto a *

Quindi:

char *a[10]; => equivale a char *(a[10]);

a è un **vettore di puntatori** a carattere.



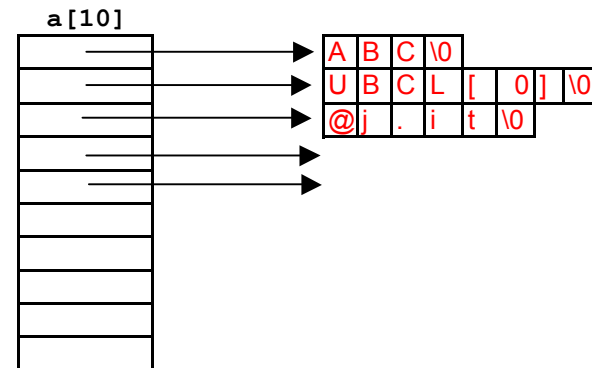
- Per un puntatore ad un vettore di caratteri è necessario forzare la precedenza (con le parentesi)

char (* a) [10];

Precisazione

Le parentesi tonde variano le precedenze
siccome `[]` hanno la precedenza rispetto a `*`

`char * a [10];` equivale a `char * (a[10]);`
è un array `a[10]` di puntatori a caratteri `char *`



Per ottenere un altro effetto invece

`char (* a) [10]` `*a` è un puntatore ad un vettore di caratteri



Arrays of Pointers

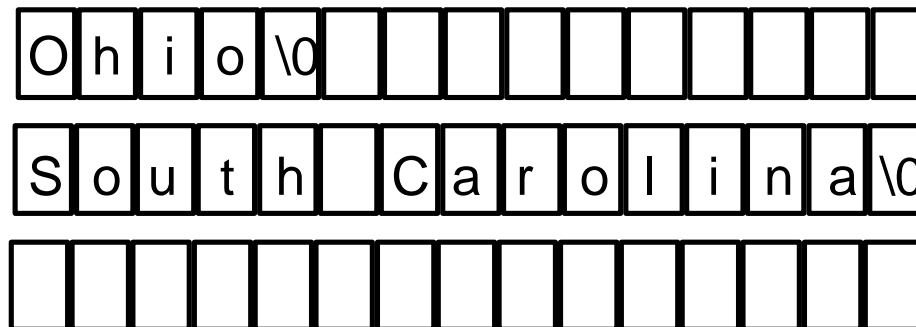
“Smooth” arrays

use same amount of storage for each string

wastes space for smaller strings

`char states[50][15] = {"Ohio", "South Carolina", ...};`

`states[0]` would reference Ohio, 15 elements used



Arrays of Pointers

“Ragged” arrays

use only amount of storage needed for each string, plus storage needed for pointers

use array of pointers

do not “waste” space for smaller strings

```
char *states2[50] = {"Ohio", "South Carolina", ...};
```

states2[0] would reference Ohio, 5 elements used,

