U.D. 6

Il costruttore di tipi complessi

1. Il Puntatore

Fonti:

Antola Dispense del Corso + A. Ciampolini – UNI_BO – Lucidi del corso

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 */
```

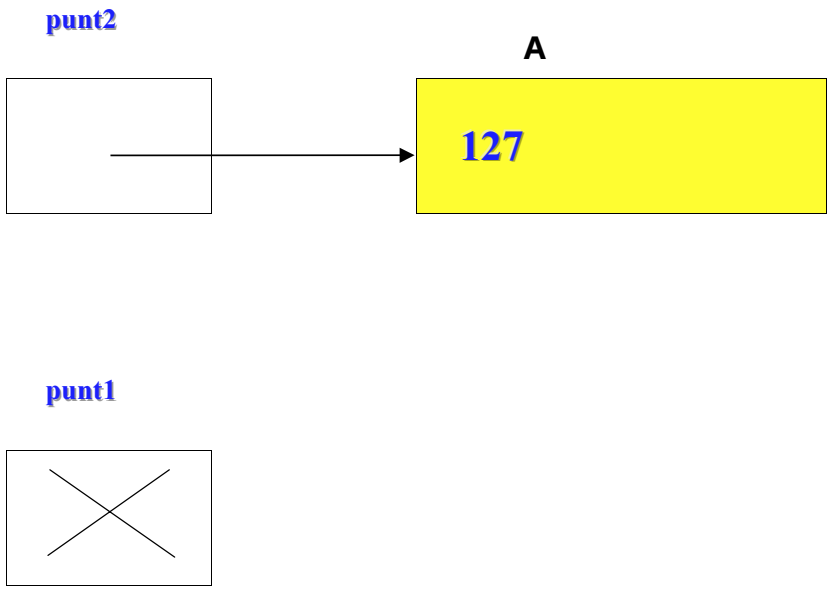
Il puntatore

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;
```

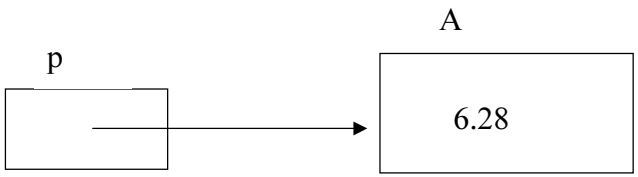
Operatore Indirizzo &:

- **&** si applica solo ad ***oggetti che esistono in memoria*** (quindi, gia` definiti).
- **& non e` 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> e` il tipo della variabile puntata
- **<**NomePuntatore> e` 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;

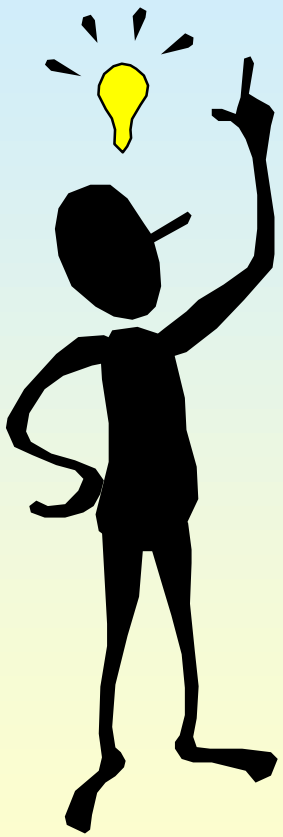
```
int *p, A;
record *q, X;

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



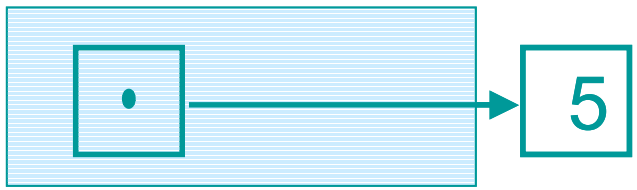
int *iptr;

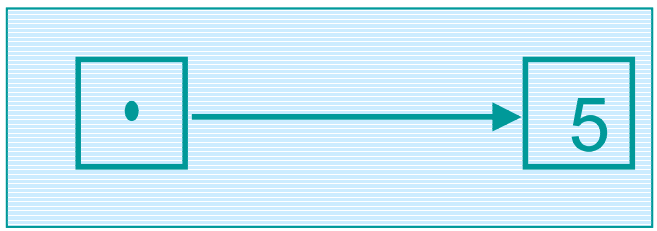
iptr di tipo int *

*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 */
```

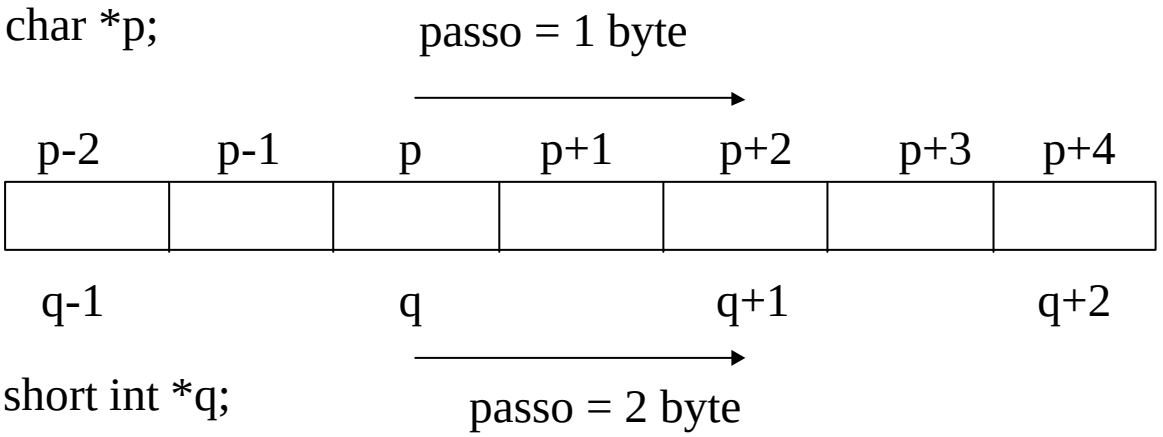
Aritmetica degli indirizzi

Si possono fare operazioni aritmetiche intere con i puntatori, ottenendo come risultato di far avanzare o riportare indietro il puntatore nella memoria, cioè di farlo puntare ad una locazione di memoria diversa.
Ovvero con i puntatori è possibile utilizzare due operatori aritmetici + e - , ed ovviamente anche ++ e --.

Il risultato numerico di un'operazione aritmetica su un puntatore è diverso a seconda del tipo di puntatore, o meglio a seconda delle dimensioni del tipo di dato a cui il puntatore punta. Questo perchè **il compilatore interpreta diversamente la stessa istruzione p++ a seconda del tipo di dato**, in modo da ottenere il comportamento seguente:

- **Sommare un'unità ad un puntatore significa spostare in avanti in memoria il puntatore di un numero di byte corrispondenti alle dimensioni del dato puntato dal puntatore.**

Ovvero se p è un puntatore di tipo puntatore a char, char *p; poichè il char ha dimensione 1, l'istruzione p++ aumenta effettivamente di un'unita il valore del puntatore p, che punterà al successivo byte.
Invece se p è un puntatore di tipo puntatore a short int, short int *p; poichè lo short int ha dimensione 2 byte, l'istruzione p++ aumenterà effettivamente di 2 il valore del puntatore p, che punterà allo short int successivo a quello attuale.



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;
  ```

  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;
```

Accesso ai campi della struttura: le notazioni sono equivalenti

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

P -> data_di_nascita.giorno
P -> cognome[0]
```

Precedenze

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

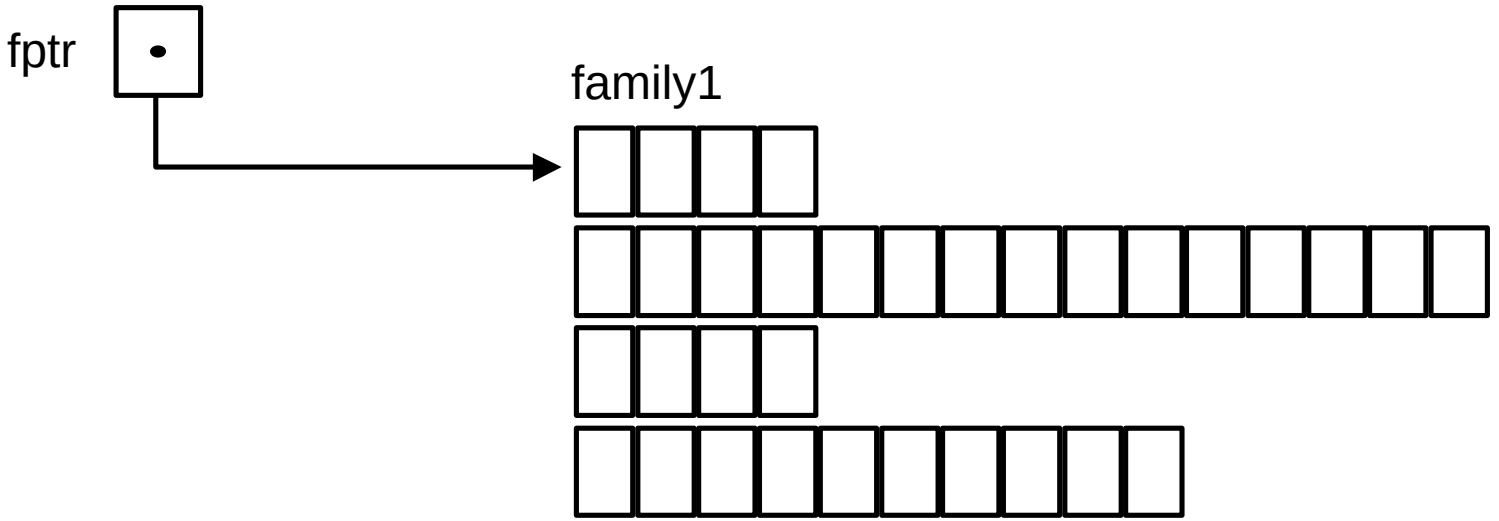
MS 2300, 2550 Beckleymeade Ave., Dallas, TX 75237

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;
```

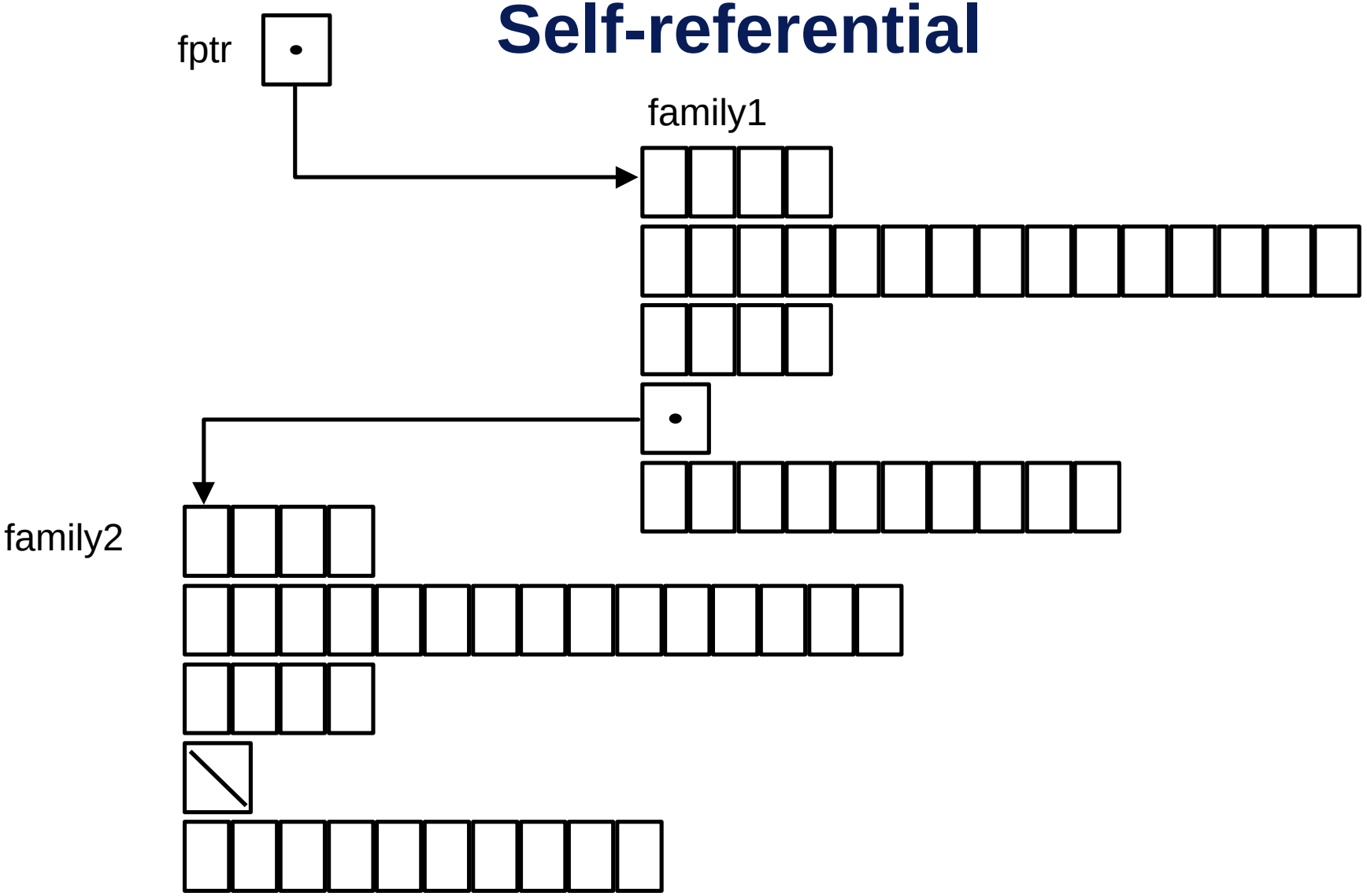


Self-referential Structs

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;
```

Self-referential

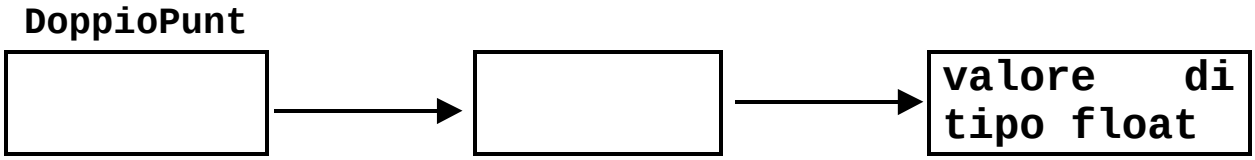


DOPPIO PUNTATORE

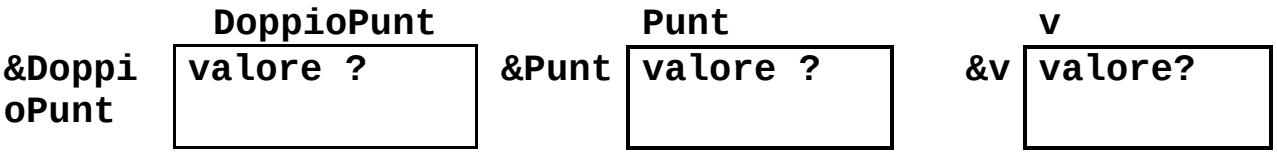
```
float      **DoppioPunt;
float      *Punt;
float      v;
```

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

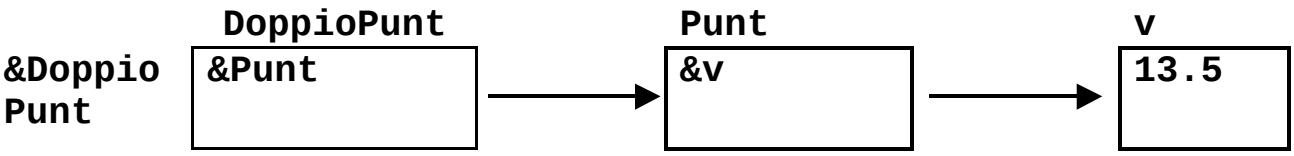


Dopo la dichiarazione:




```
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;

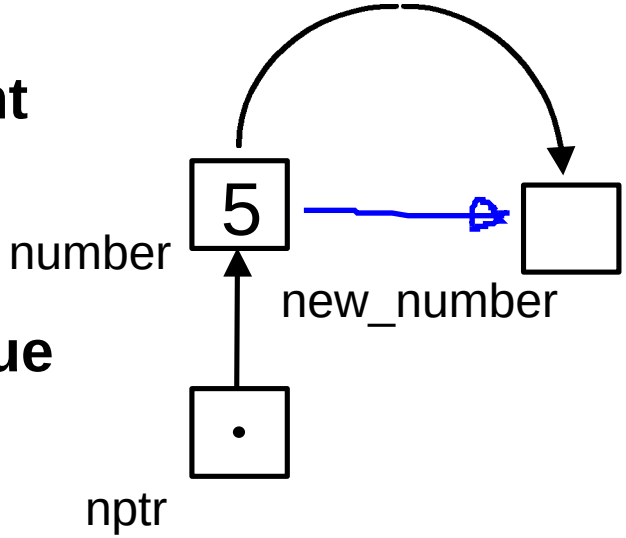
- **dereference pointer to "reach" value indirectly**

  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**

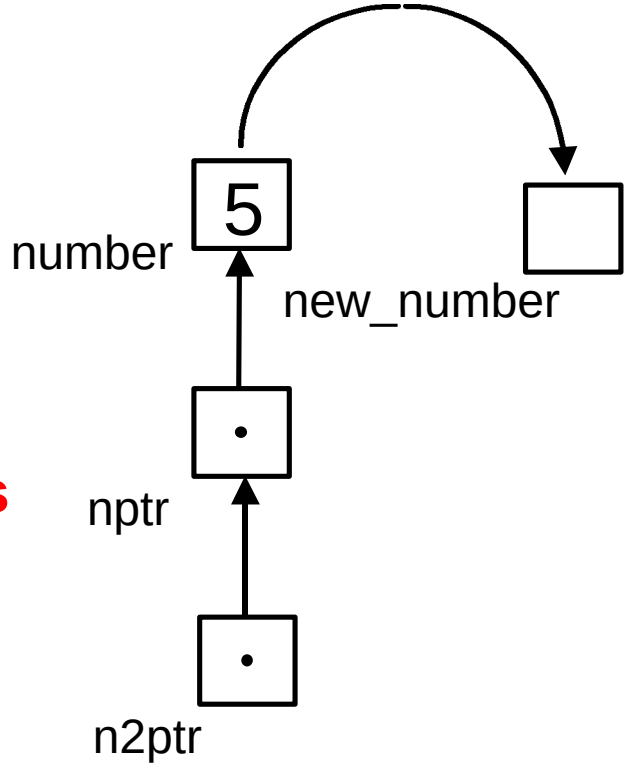
  new_number = **n2ptr;

- **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** (cioe` 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;  /* p 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 e' 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.

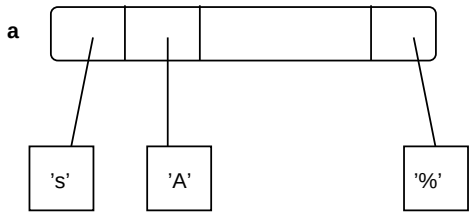
Vettori & Puntatori

☞ **[ ]** 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];

Arrays of Pointers

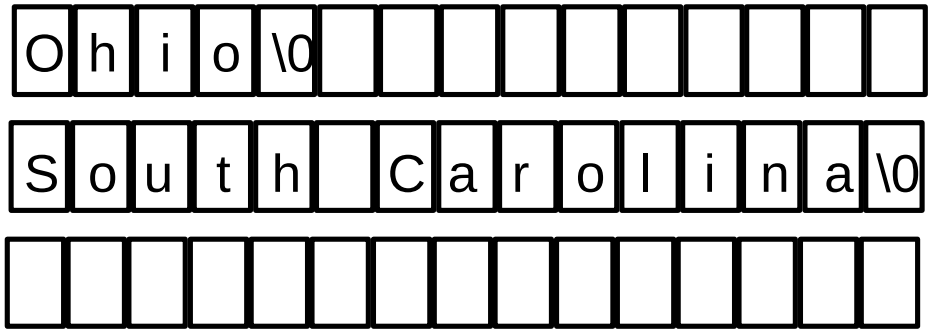
“Smooth” arrays

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

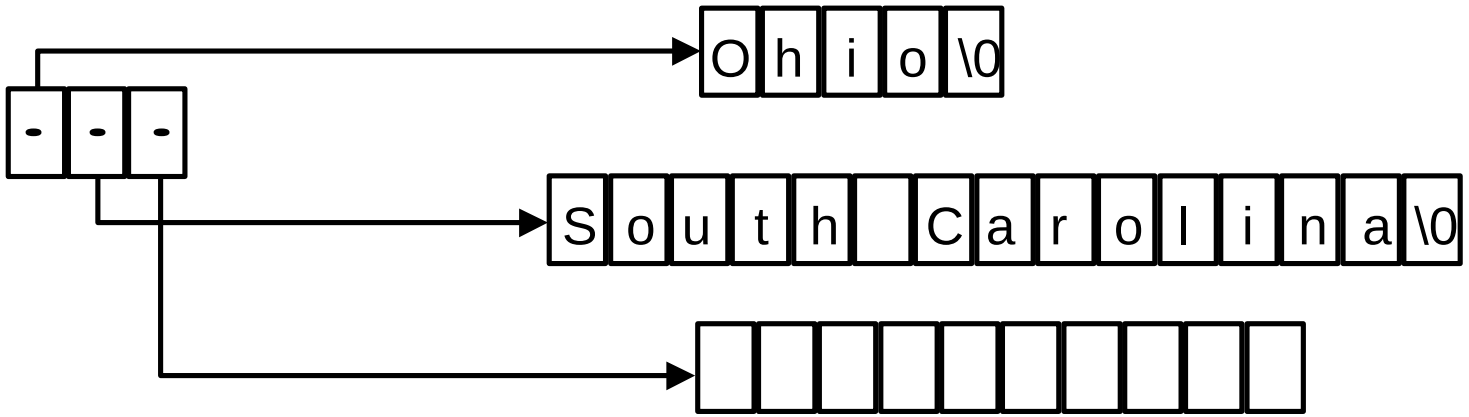
use only amount of storage needed for each string, plus storage needed for pointers

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,



Uso di Puntatori

```
int X,Y,Z;
int V[5]={ 20,21,22,23,24 }  ;
int A, *P, *Q, **PP;
```

Indir		nome	
1000		X	X=10;
1002		Y	Y=X+1;
1004		Z	Z=X+Y;
1006	20	V[0]	
1008	21	V[1]	
1010	22	V[2]	
1012	23	V[3]	
1014	24	V[4]	
1016		A	V[5}=50;
1018		P	P=V;
1020		Q	Q=&V[4];
1022		PP	PP=&Q;
1024			

Indir		nome	
1000	10	X	
1002	11	Y	
1004	21	Z	
1006	20	V[0]	
1008	21	V[1]	
1010	22	V[2]	
1012	23	V[3]	
1014	24	V[4]	
1016	50	A	
1018	1006	P	*P=57;
1020	1014	Q	A=*Q+10;
1022	1020	PP	X=**PP+100;
1024			

Indir		nome
1000	124	X
1002	11	Y
1004	21	Z
1006	57	V[0]
1008	21	V[1]
1010	22	V[2]
1012	23	V[3]
1014	24	V[4]
1016	34	A
1018	1006	P
1020	1014	Q
1022	1020	PP
1024		

Pointer Types

- pointer variable definitions:

```
int *ip;                    // pointer to variable of type int
char *c1, *c2, *c3;         // pointer to variables of type char
float **fp;                 // pointer to pointer to variable of type float
```

- dereference operator (*) and address-of operator(&):

```
int i, *ip = &i, **ipp = &ip, ***ippp = &ipp;
// i is a variable of type int
// ip is a pointer to a variable of type int
// ipp is a pointer to a pointer to a variable of type int
// ippp is a pointer to a pointer to a pointer to a variable of type int
double d = 2.7183, *dp = &d;
// the value of d is 2.7183
// the value of &d is an address (where 2.7183 is stored...)
// the value of dp is the address of d.
// the value of *dp is the value of what dp points to (d or 2.7183)
```

- string pointers:

```
char *string = “This is a string.\n”;
// strings are null (‘\0’) terminated by convention
char  *same_string = string;
```

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

Variabili Dinamiche

In C si possono definire e` possibile classificare le variabili in base al loro tempo di vita; e` possibile individuare due categorie:

- variabili **automatiche**
- variabili **dinamiche**

Variabili automatiche:

- L'allocazione e la deallocazione di variabili automatiche e` effettuata automaticamente dal sistema (senza l'intervento del programmatore).
- Ogni variabile automatica ha un nome, attraverso il quale la si puo` riferire.
- Il programmatore non ha la possibilita` di influire sul tempo di vita di variabili automatiche.

Variabili Dinamiche

Variabili dinamiche:

- Le variabili *dinamiche* devono essere allocate e deallocate esplicitamente dal programmatore.
- L'area di memoria in cui vengono allocate le variabili dinamiche si chiama *heap*.
- Le variabili dinamiche non hanno un identificatore associato ad esse, ma possono essere riferite soltanto attraverso il loro indirizzo (mediante i puntatori).
- Il tempo di vita delle variabili dinamiche e` l'intervallo di tempo che intercorre l'allocazione e la deallocazione (che sono stabilite dal programmatore).

☞ tutte le variabili viste finora rientrano nella categoria delle **variabili automatiche**.

Variabili Dinamiche

☞ Il C prevede funzioni standard di **allocazione deallocazione** per variabili dinamiche:

- malloc
- free

Non sono definite a livello di linguaggio di programmazione, ma a **livello di sistema operativo**, mediante la libreria standard **<stdlib.h>.**

Variabili Dinamiche

Allocazione di variabili dinamiche:

La memoria dinamica viene allocata con la funzione standard ***malloc***:

punt = (tipodato *) malloc **(sizeof (tipodato));**

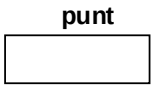
- **tipodato** e` il tipo della variabile puntata
- **punt** e` una variabile di tipo **tipodato ***
- **sizeof**() e` una funzione standard che calcola il numero di bytes che occupa il dato specificato come argomento
- e` necessario convertire esplicitamente il tipo del valore ritornato (casting): (tipodato *) malloc(..)

Significato:

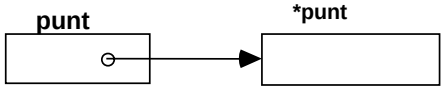
☞ La malloc provoca la creazione di una variabile dinamica nell'***heap*** e restituisce come valore l'indirizzo della variabile creata.

Ad esempio:

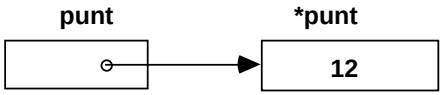
```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
```




```
punt=(tp )malloc(sizeof(int));
```




```
*punt=12
```



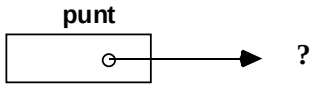
Variabili dinamiche

Deallocazione:

Si rilascia la memoria allocata dinamicamente con:

free (punt);

dove punt e` l'indirizzo della variabile da deallocare.



Dopo questa operazione, la cella di memoria occupata da *punt viene deallocata: *punt non esiste piu`.

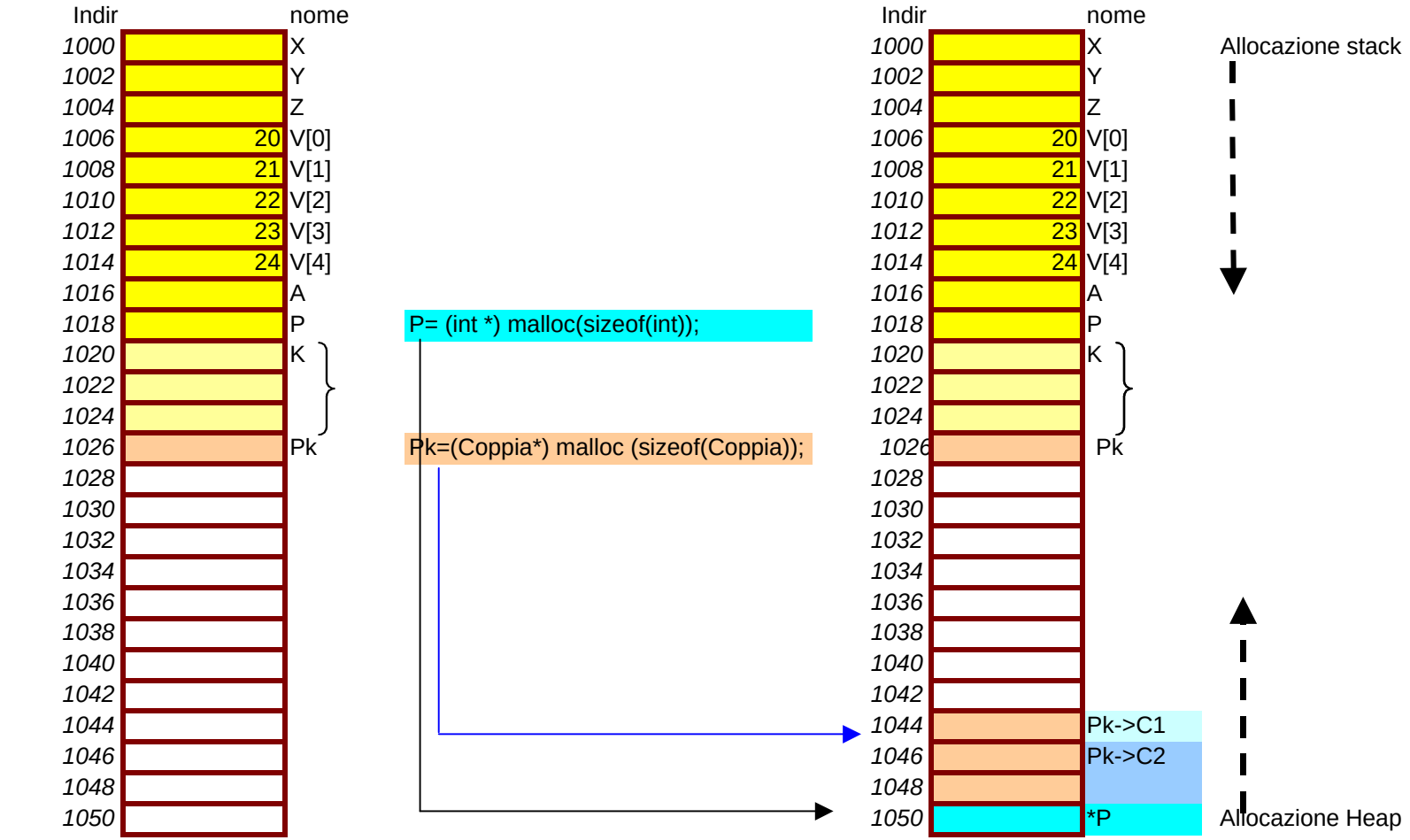
Esempio:

```
main()
{
 char A, *p;

 A='Z';
 p=(char *)malloc(sizeof(char));
 *p=A;
 ...
 <uso di *p>
 ...
 free(p);
}
```


Esempio:

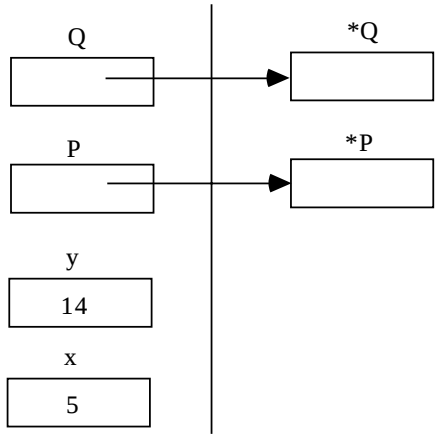
```
#include <stdlib.h>
main()
{
int *p;
  /*definizione del puntatore p
    ad intero;il contenuto di p non è
    ancora definito */

p = (int *) malloc(sizeof (int));
  /*definizione del contenuto di p:
    indirizzo di una cella di memoria
    allocata dinamicamente*/

*p = 55;
   /* assegnamento di un valore alla
      cella *p referenziata da p */

free(p);
   /* deallocazione della cella
      referenziata da p; il contenuto
      di p non è più definito */
}
```

```
int X,Y,Z;
int V[5]={ 20,21,22,23,24 }  ;
int A, *P, *Q, **PP;
typedef  struct { int C1; float C2 } Coppia;
Coppia K, *Pk;
```

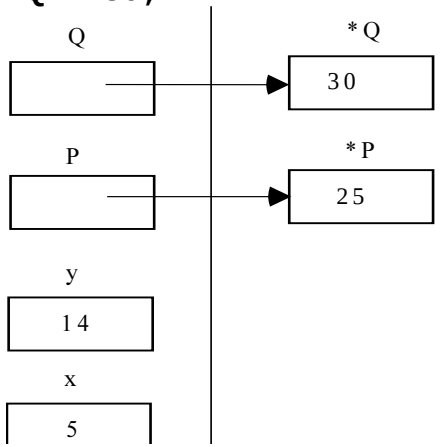


Esempio:

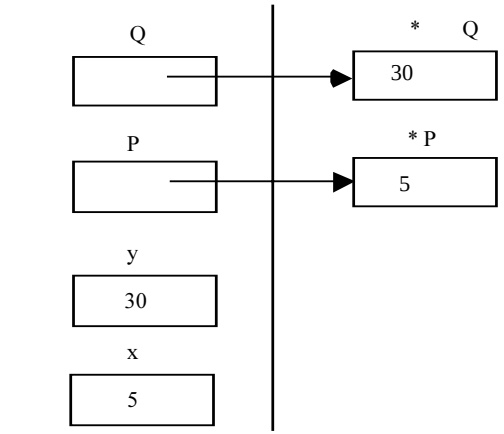
```
main()
{
int *P, *Q, x, y;

x=5;
y=14;
P=(int *)malloc(sizeof(int));
Q=(int *)malloc(sizeof(int));
```



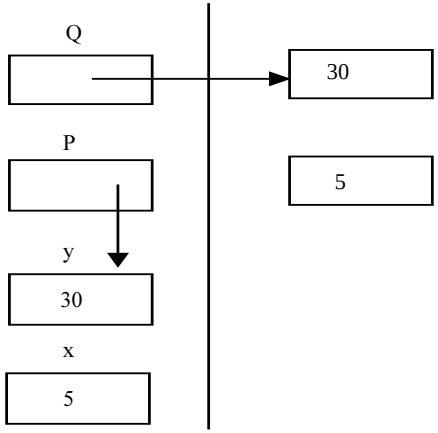

```
*P = 25;
*Q = 30;
```




```
*P = x;
y = *Q;
```




```
P = &y;
```




```
...
}
```

☞ l'ultimo assegnamento ha come effetto collaterale la perdita dell'indirizzo di una variabile dinamica (quella precedentemente referenziata da P) che rimane allocata ma non é più utilizzabile!

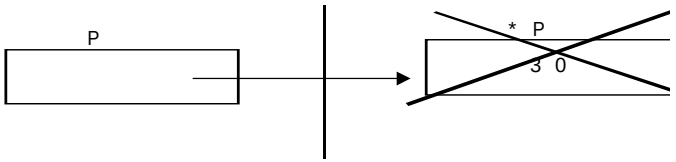
Problemi legati all'uso dei Puntatori

Riferimenti pendenti (dangling references):

Possibilità di fare riferimento ad aree di memoria non più allocate.

Ad esempio:

```
int *P;
P = (int *) malloc(sizeof(int));
...
free(P);
*P = 100;    /*  Da non fare!  */
```



Problemi legati all'uso dei Puntatori

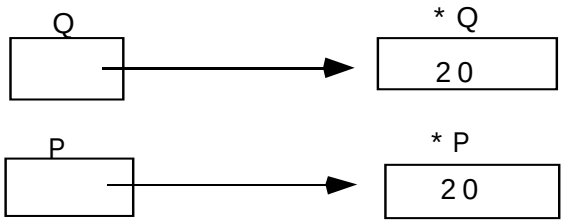
Aree inutilizzabili:

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

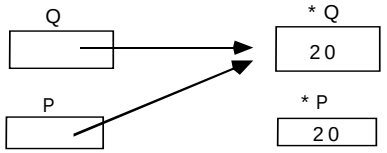
Ad esempio:

```
int *P,*Q;
  P = (int *) malloc ( sizeof (int));
  Q = (int *) malloc ( sizeof (int));
  *P = 30;    *Q = 20;
```

***P = *Q;**



P = Q;



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

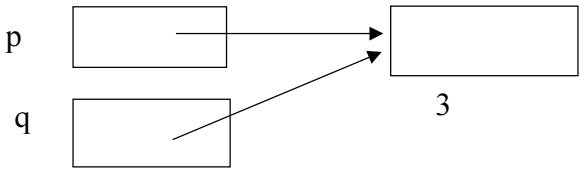
Problemi legati all'uso dei Puntatori

Aliasing:

Possibilita` di riferire la stessa variabile con puntatori diversi:

Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
       variabile */
```




```
*q = 10;  /* anche *p e' cambiato! */
```

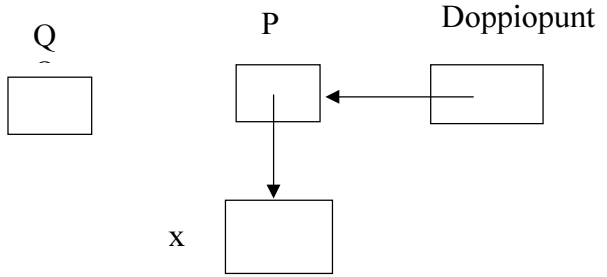

Puntatori a puntatori (handle)

Un puntatore pu` puntare a variabili di tipo qualunque (semplici o strutturate); puo` puntare anche a un puntatore:

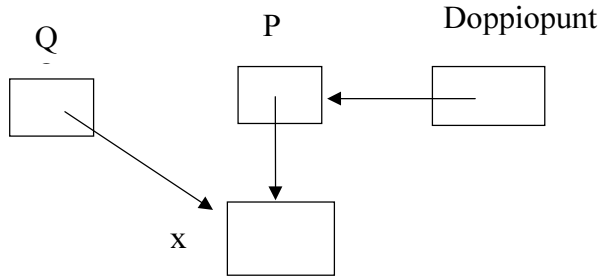
[typedef] **TipoDato** ****TipoPunt;**

Ad esempio:

```
int x, *P, *Q, **DoppioPunt;
 P = &x;
 DoppioPunt = &P;
```




```
Q = *DoppioPunt;
```



Pointers

- Every variable—even a `const`—has an address ("lvalue") and value ("rvalue"), e.g., (which used where?): `int a = 4; a ++;`

- Definition, initialization and assignment:

```
int  some_info, *in_ptr = &some_info, **int_buffer_ptr;
char ch, *char_index;    // allocation for _any_ pointer:
char_index = &ch;        //   (sizeof (int)) bytes (why?)
```

  note "`*`" in initialization, but not in assignment (why?)

- Dereferencing and pointer arithmetic:

```
*in_ptr += 2;    // == some_info += 2;
in_ptr += 2;    // == in_ptr increases by 2 * sizeof (int);
```

- Dereferencing $\Rightarrow$ point to something or set to 0 *at all times*

  Primarily for free store, unnamed memory allocation.

Constant Pointers

```
float *fp;
const float *cfp;    // cfp points to a "const float"
// fcp is a const pointer, to a float
float *const fcp = &some_float;
const float *const cfcp = &some_const_float;
```

	can change what it points to	can point to a `const float`	needs initialization (& cannot be changed)
`fp`	$\surd$	$\times$	$\times$
`cfp`	$\times$	$\surd$	$\times$
`fcp`	$\surd$	$\times$	$\surd$
`cfcp`	$\times$	$\surd$	$\surd$

Pointers to `const` are mainly for function arguments.

Arrays and Pointers

```
long id_numbers [5];
long *id_num_ptr = &(id_numbers [3]);
```

- Similarities
  - Both tags alone refer to address of its first element

    ```
    if (id_num_ptr == id_numbers) { /* ... */ }
    ```
  - $\Rightarrow$ both can use array and pointer syntax

    ```
    id_num_ptr [1] = *(id_numbers + 2);
    ```
- Differences
  - `id_numbers` only has an rvalue: *refers* to address of beginning of array and cannot be changed
  - `id_num_ptr` also has an lvalue: an extra `(long *)` is allocated and can be set to address of a `long`

1–D Pointer and Array Definitions

```
double *dp = 1000, da [5];
```

- Assume allocated at 4000 and 5000, resp.

- Bytes allocated
  - `dp`: `sizeof (void *)` [4]
  - `da`: `5 * sizeof (double)` [40]

- `sizeof (dp)` and `sizeof (da)` return these numbers

- Note: initialization to 1000 is a bad idea (why?); better:
  - `dp` gets set to address of `double`, e.g., `&(da [3])`
  - `dp` gets return value of `new()` (later)

1–D Pointers

syntax	type	lvalue	rvalue	comments
`dp`	`double *`	4000	1000	name alone
`dp + 2`	`double *`	—	1016	pointer arithmetic
Rule: in bytes: `dp + 2 * sizeof (double)` (i.e., remove 1 `*`)				
`*dp`	`double`	1000	$r(1000)^*$	pointer dereferencing
Rule: add a `*` of dereferencing $\Rightarrow$ drop a `*` of the type				
`dp [3]`	`double`	1024	$r(1024)^*$	array syntax
Rule: `foo [n]` is just `*(foo + n)` (a combination)				

$^*r(n)$ means whatever resident at memory location n

1–D Arrays

syntax	type	lvalue	rvalue	comments
`da`	`double []`	—	5000	name alone
Rule: array without `[]` is *rvalue* of array beginning address				
Rule: `double []` is *like* a `double *` (but not exactly)				
..... same as pointer				

- Note: the size of 5 `doubles` was *only* used for
  - initial allocation
  - future invocations of `sizeof()`

Now for 2–D arrays ....

2–D Pointer and Array Definitions

```
double **dpp = 2000, dm [7][11];
```

- Assume allocated at 6000 and 7000, resp.
- Bytes allocated
  - `dpp`: `sizeof (void *)` [4]
  - `dm`: `7 * 11 * sizeof (double)` [616]
- `sizeof (dpp)` and `sizeof (dm)` return these numbers
- Again, initialization to 2000 is a bad idea
- Now for same analysis, with (almost) same rules

2–D Pointers

syntax	type	lvalue	rvalue	comments
`dpp`	`double **`	6000	2000	name alone
`dpp + 2`	`double **`	—	2008	pointer arithmetic
`*dpp`	`double *`	2000	$r(2000)^*$	pointer dereferencing
`**dpp`	`double`	9788	$r(9788)$	multiple dereferencing
`dpp [3]`	`double *`	2012	$r(2012)^+$	array syntax
Rule: `foo [n]` is just `*(foo + n)`				
`dpp [3][5]`	`double`	8884	$r(8884)$	multi-array syntax
Rule: `foo [n][m]` is just `*(*(foo + n) + m)`				

*say, 9788; +say, 8844

2–D Arrays

syntax	type	lvalue	rvalue	comments
`dm`	`double [][11]`	—	7000	name alone
Rule: compiler needs to know how to jump, e.g., `dm [1][0]`				
Rule: array type retains all dimensions except first				
`dm + 2`*	`double [][11]`	—	7176	partial pointer syntax
Rule: `dm + n` is just rvalue of `&(dm [n][0])`				
`dm [2]`*	`double []`	—	7176	partial array syntax
Rule: `dm [n]` is just rvalue of `&(dm [n][0])`				
`dm [2][4]`	`double`	7208	$r(7208)$	full array syntax

*note the same values, but different types (and ∴ arithmetic)