puntatori

Attributi di un “oggetto”

nome o identificatore;

tipo;

valore (o valori);

indirizzo;

Indirizzo

Consideriamo la dichiarazione con inizializzazione:

```
int  numero=10;
```

Questa definisce una variabile con attributi

identificatore: numero

tipo: int

valore: 10

indirizzo: &numero, noto solo dopo che il codice compilato sia stato caricato in memoria

Puntatori

- I puntatori sono variabili di tipo int usate per memorizzare gli indirizzi di altre variabili o strutture dati (che possono essere a loro volta interi ma anche char, double...).

- I puntatori vengono utilizzati
  - per velocizzare l'accesso ai dati in memoria (ad esempio la gestione di array di grandi dimensioni)
  - per allocare e liberare memoria durante la fase di esecuzione del programma, ovvero dinamicamente
  - per condividere informazioni tra diversi blocchi (funzioni) di un programma senza duplicare in memoria strutture dati di grandi dimensioni.

Dichiarazione di un puntatore

E' necessario specificare il tipo cui punta la variabile puntatore.

Un puntatore p_int ad una locazione di memoria contenente una variabile intera si definisce con:

```
int  *p_int;
```

Un puntatore p_double ad una locazione di memoria contenente una variabile double si definisce con:

```
double  *p_double;
```

etc...

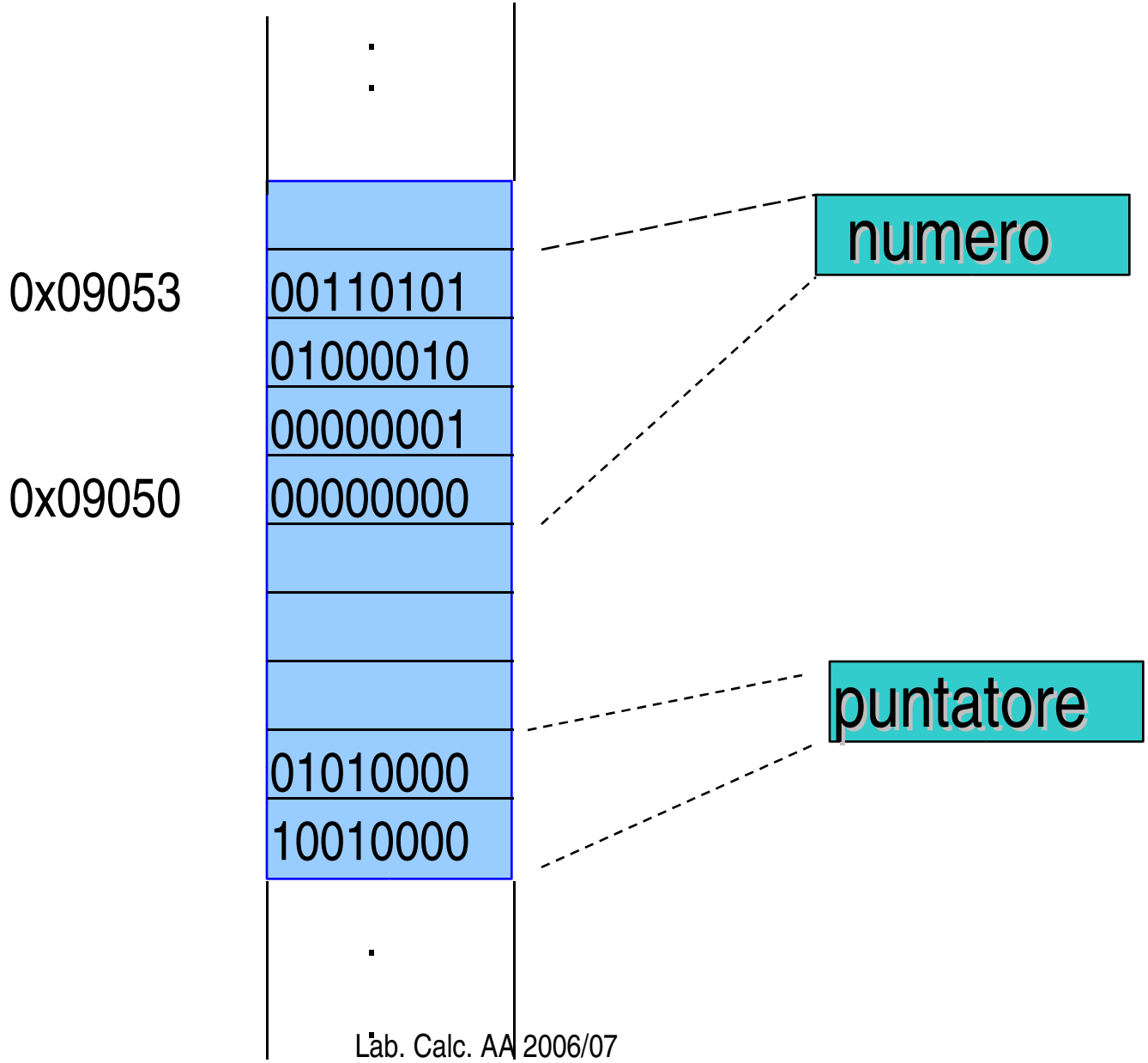
La posizione di * è irrilevante, posso scrivere anche

```
tipo   * p;
tipo*   p;
```

attenzione: finche' non viene inizializzato ad un indirizzo valido il puntatore non deve essere usato!

Uso dei puntatori

```
    int  numero;
/* il compilatore riserva spazio in memoria per un intero ad un
     certo indirizzo, ad esempio 0x09050 */
    numero = 82485;
/* un intero a 32 bit che in binario si scrive
   00000000 00000001 01000010 00110101 */
    int *puntatore;
/* puntatore ad un oggetto intero non precisato */
     puntatore = &numero;
/* puntatore a numero (posto uguale all'indirizzo di numero)
     mediante l'operatore unario di indirizzamento & */
    printf("%p\n",puntatore);
/* stampa il contenuto di puntatore in formato esadecimale, ovvero
     l'indirizzo di numero */
```


0x09053
0x09050
00110101
01000010
00000001
00000000
01010000
10010000
numero
puntatore

se ora poniamo

numero=0;

e stampiamo nuovamente il valore di puntatore questo non sara' cambiato.

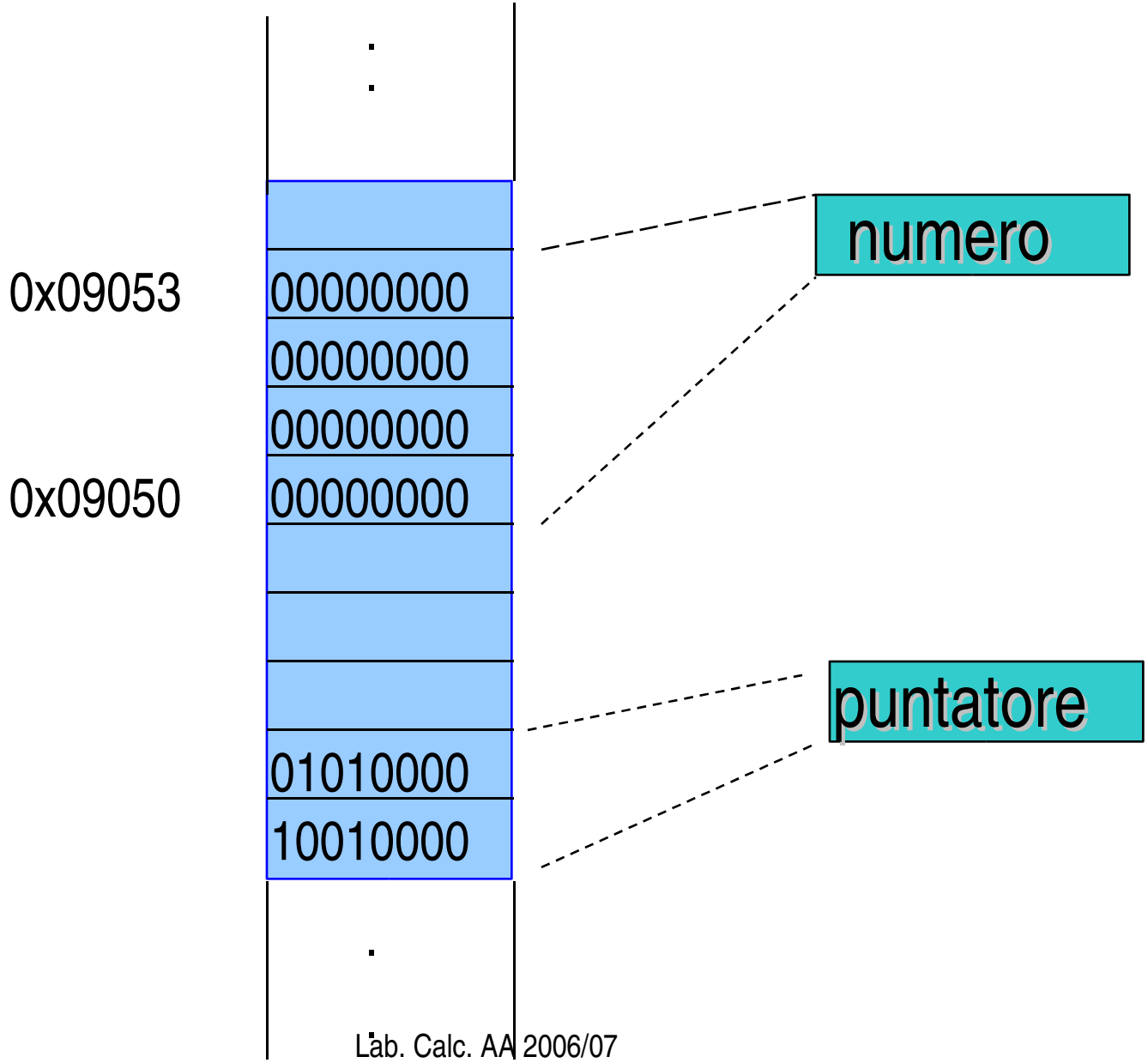
Per accedere o modificare il valore della variabile puntata si utilizza l'operatore unario di indirezione *:

*puntatore=0;

e' equivalente a

numero=0;

Porre una variabile uguale a *puntatore o a numero fornisce lo stesso risultato.


0x09053
0x09050
00000000
00000000
00000000
00000000
01010000
10010000
numero
puntatore

printf/scanf

- ora che sappiamo che &v rappresenta l'indirizzo di memoria della variabile v possiamo notare che
  - printf deve solo leggere il valore della variabile e mandarla in stampa, gli basta conoscerne l'identificatore v
  - scanf deve poter modificare il contenuto di una locazione di memoria per caricarvi il dato immesso da tastiera, ha bisogno di conoscerne l'indirizzo &v

Vettori e puntatori

In C il nome di un array e' proprio il puntatore all'indirizzo di memoria che contiene il primo elemento dell'array stesso.

int a[25];	a è un puntatore a interi
double x[25];	x è un puntatore a double
char ch[25];	ch è un puntatore a char

mentre a[0], x[5] o ch[10] sono rispettivamente un intero, un double ed un char.

Poiche' gli elementi di un array occupano locazioni di memoria consecutive, le dichiarazioni

```
char st[] = "A string";
char *st="A string";
```

sono equivalenti, infatti:

st[0] ha valore 'A', *st ha valore 'A'
st[7] ha valore 'g', *(st+7) ha valore 'g'
st[8] ha valore '\0', *(st+8) ha valore '\0'

se proviamo a stampare i valori dei puntatori st e st+1 troviamo due indirizzi che distano tra loro 1 byte: infatti i char occupano 8 bit. In un vettore di int i gli indirizzi distano 2 byte, in un vettore di double distano 4 byte...

Indirizzamento

- &st[0] è l'indirizzo di st[0], ovvero è la stessa cosa di st
- &st[1] è l'indirizzo di st[1], ovvero la voce successiva a quella cui punta st, è la stessa cosa di st + 1
- &st[k] è la stessa cosa di st + k per ogni k valido

Inizializzazione

Sappiamo di poter dimensionare e inizializzare un vettore di numeri con la dichiarazione

int list[] = {2,3,4,5};

Va però osservato che non è possibile farlo mediante l'istruzione

int * plist ={2,3,4,5};

Infatti plist è un puntatore all'indirizzo da cui si iniziano a memorizzare uno o più numeri interi (nel nostro caso 2,3,4 e 5) e non un vettore di interi!

list[3] è equivalente a *(plist+3)

esempi e errori (Q6.1 del testo Barone et al.)

```
int a;
double b, *pd;
pd=&a;
```

questo codice produce un errore di compilazione:

```
cannot convert 'int*'  to 'double*' in assignment
```

```
double b, c, *pd;
b=3.14;
c=&(*pd);
```

questo codice produce un errore di compilazione:

```
cannot convert 'double*'  to 'double' in assignment
```

```
double b,*pd;
pd=&b;
*pd=3.14;
printf("%lf\n",b);
```

questo codice funziona e stampa 3.140000

```
double b, c,*pd;
b=3.14;
pd=&c;
*pd=&b;
```


questo codice produce un errore di compilazione:

```
cannot convert 'double*'  to 'double' in assignment
```

```
int a;
double b=3.14,*pd;
pd=&b;
a=*pd;
```


questo codice genera un warning:

```
warning: assignment to `int' from `double'
```

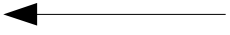
```
double b, c,*pd;
b=3.14;
pd=&c;
*pd=b;
```

questo codice funziona e stampa 3.140000

```
double b, *pd;
 *pd=3.14;
 b=*pd;
 printf("%lf\n",b);
```
questo codice funziona e stampa 3.140000

```
double b, *pd;
 *pd=3.14;
 &b=pd;
```


questo codice produce un errore di compilazione:
`non-lvalue in assignment`
&b non e' una variabile cui si possa assegnare un valore (lvalue) ma un indirizzo definito dal sistema

```
 int a;
 a=&2;
```


questo codice produce un errore di compilazione:
`non-lvalue in unary `&'`
ovvero non si puo' applicare l'operatore unario di indirizzamento & ad un argomento che non sia una variabile (lvalue = valore che puo' trovarsi a sinistra del segno = in un'assegnazione)

I/O da file

Per file formattati si usano fprintf e fscanf, come nel seguente esempio (6.11 del testo):

```
#include <stdio.h>
#include <stdlib.h>
#define LEN 100
main () {
  FILE * fp;
  int i, k;
  double x, data[LEN];
  if ((fp = fopen("random.dat", "w")) == NULL ) {
      printf("Errore nell'apertura del file
random.dat\n");
      exit(EXIT_FAILURE);
  }
  for (i = 1; i <= LEN; i++) {
    x = (double)rand() / (double)RAND_MAX;
    fprintf(fp, "%d %f\n", i, x);
  }
  fclose(fp);
```

```
  fp = fopen("random.dat", "r+");
  for (i = 1; i <= LEN; i++) {
    fscanf(fp, "%d %lf", &k, &x);
    *(data + LEN - k) = x;
  }
  fprintf(fp, "\n INVERSIONE \n");
  for (i = 1; i <= LEN; i++) {
    fprintf(fp, "%d %f\n", i, *(data + i - 1));
  }
  fclose(fp);
}
```

per file binari (da aprire con fopen e le opzioni “wb” o “rb”) si usano invece le funzioni fwrite e fread che agiscono copiando una sequenza di locazioni di memoria (vedere testo per maggiori dettagli).

prenotazione dinamica della memoria
(testo par. 10.2.2)

- la memoria riservata puo' essere riservata
  - staticamente: in fase di compilazione
  - dinamicamente: in fase di esecuzione del programma
- vantaggi dell'allocazione dinamica
  - scelta della dimensione degli array in fase di esecuzione
  - lo spazio riservato non si libera automaticamente
- svantaggi
  - lentezza
  - responsabilita' del programmatore
    - nel verificare che la memoria richiesta sia effettivamente disponibile
    - nel liberare la memoria non piu' necessaria

sintassi

```
void *malloc(size_t n);
```

- restituisce un puntatore a n byte di spazio di memoria on inizializzato oppure NULL se lo spazio non e' disponibile
- il puntatore e' a un void (nessun tipo) e non puo' essere usato se non si specifica il tipo di dati che vogliamo salvare nella memoria riservata mediante un *cast* esplicito

```
float *xFloat;
xFloat = (float *)malloc(100*sizeof(float));
```

nota: size_t e' un tipo di intero senza segno definito in <stdef.h>

sintassi

- per inizializzare a zero la memoria riservata usare calloc invece di malloc

```
xFloat = (float *)calloc(100, sizeof(float));
```

- per liberare la memoria usare la funzione free con argomento il puntatore all'area di memoria interessata

```
free(xFloat);
```

- per ampliare uno spazio di memoria gia' riservato (ottenendo delle locazioni contigue alle precedenti fino al raggiungimento di n byte)

```
xFloat2=realloc(xFloat, n);
```

attenzione, in questo caso la memoria aggiuntiva non e' inizializzata a 0

Regole

- Il programma deve contenere una chiamata alla funzione free per ogni puntatore inizializzato ad aree di memoria ottenute dinamicamente.
- free va chiamata alla fine del blocco di codice che ha allocato la memoria, in un punto ben visibile, per essere sicuri di non utilizzare più quel puntatore.