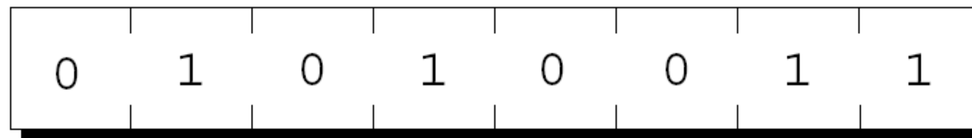
Capitolo 11

Puntatori

Variabili “puntatore”

- Cerchiamo di capire come le informazioni sono memorizzate nella memoria del computer
- La memoria è un sequenza di ***byte***
- Ogni byte è composto da 8 bit:



- Ogni byte ha un ***indirizzo***

Puntatori

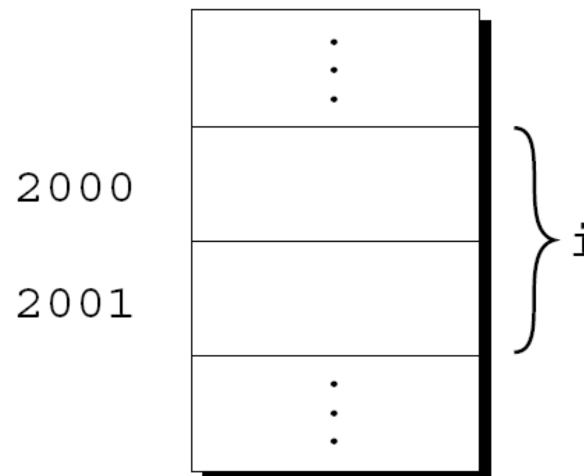
- Se la memoria è composta da n byte allora possiamo pensare agli indirizzi come ai numeri da 0 a $n - 1$:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
n-1	01000011

- Esempio indirizzo:
  - 0x6CE67A4E

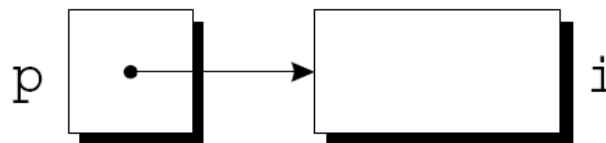
Puntatori

- Ogni variabile occupa una o più locazioni di memoria (byte)
- L'indirizzo del primo byte viene usato per rappresentare la variabile: è l'indirizzo della variabile
- Nella figura, l'indirizzo della variabile `i` è 2000:



Puntatori

- Gli indirizzi delle locazioni di memoria possono essere memorizzati in variabili speciali dette ***puntatori***
- Quando memorizziamo l'indirizzo di una variabile `i` nel puntatore `p`, diciamo che `p` "punta a" `i`
- Ecco una rappresentazione grafica:



- In pratica, usando l'esempio precedente, risulta `p=2000`

Dichiarazione dei puntatori

- Quando una variabile puntatore viene dichiarata, il suo nome deve essere preceduto da un asterisco:

  `int *p;`
- `p` è una variabile che può contenere l'indirizzo di un ***oggetto*** di tipo `int`.
- Usiamo il termine *oggetto* al posto di *variabile* in quanto `p` potrebbe puntare ad una zona di memoria che non appartiene ad una variabile

Dichiarazione dei puntatori

- Variabili puntatore possono stare insieme ad altre variabili nella dichiarazione:

```
int i, j, a[10], b[20], *p, *q;
```

- Il C richiede che ogni puntatore punti **solo** ad oggetti del tipo specificato nella dichiarazione (detto il tipo ***referenziato*** dal puntatore):

```
int *p;      /* points only to integers   */
double *q;   /* points only to doubles    */
char *r;     /* points only to characters */
```

- Il tipo referenziato può essere qualsiasi tipo

Indirizzi e operatori di indirizzamento

- Il C fornisce una coppia di operatori specifici per i puntatori
- Per indicare l'indirizzo di una variabile si può usare ***l'operatore indirizzo*** &
- Per accedere all'oggetto puntato da un puntatore si può utilizzare ***l'operatore di indirizzamento*** *
  - NOTA: l'asterisco della dichiarazione NON è un operatore di indirizzamento

L'operatore indirizzo

- La dichiarazione di una variabile puntatore crea la variabile puntatore senza che essa punti ad un oggetto

```
int *p;  /* points nowhere in particular */
```

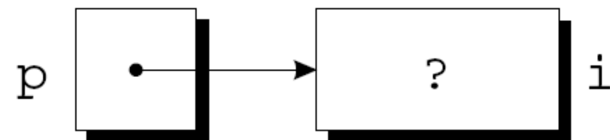
- Per utilizzare il puntatore `p` dobbiamo prima assegnargli un valore

L'operatore di indirizzo

- Un modo per assegnare un indirizzo ad un puntatore sfrutta l'operatore indirizzo:

```
int i, *p;
…
p = &i;
```

- Assegnare a `p` l'indirizzo di `i` significa far sì che che `p` punti ad `i`:



L’operatore indirizzo

- È possibile inizializzare un puntatore nel momento della dichiarazione:

```
int i;
int *p = &i;
```

- La dichiarazione di `i` può essere combinata con quella di `p`:

```
int i, *p = &i;
```

L'operatore di indirizzamento

- Una volta che un puntatore punta ad un oggetto è possibile usare l'operatore di indirizzamento `*` per accedere a ciò che è memorizzato nell'oggetto
- Se `p` punta ad `i`, possiamo stampare il valore di `i` in questo modo:

```
printf("%d\n", *p);
```

- Applicando `&` ad una variabile si ottiene l'indirizzo della variabile; applicando `*` ad un puntatore si ottiene il valore della variabile:

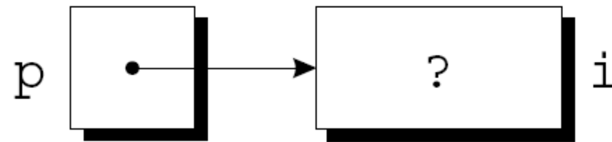
```
j = *&i;    /* same as j = i; */
```

L’operatore di indirizzamento

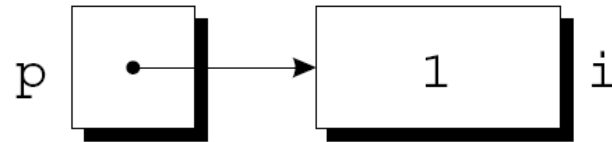
- Quando `p` punta ad `i`, `*p` è un ***sinonimo (alias)*** di `i`.
  - `*p` ha lo stesso valore di `i`.
  - Cambiando il valore di `*p` si cambia il valore `i`.
- L’esempio nella prossima slide illustra l’equivalenza tra `*p` ed `i`.

L’operatore di indirizzamento

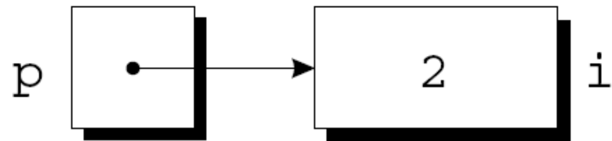
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

L’operatore di indirizzamento

- Se si applica l’operatore di indirizzamento ad un puntatore non inizializzato si ottiene un valore non predicibile

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assegnare un valore a `*p` è pericoloso se `p` non è inizializzato. Nel caso migliore si otterrà un errore durante l’esecuzione.

```
int *p;
*p = 1;    /*** WRONG ***/
```

Assegnamento ai puntatori

- Il C permette di usare l'operatore di assegnamento per copiare puntatori dello stesso tipo

- Assumendo di aver fatto la dichiarazione:

```
int i, j, *p, *q;
```

- Possiamo assegnare

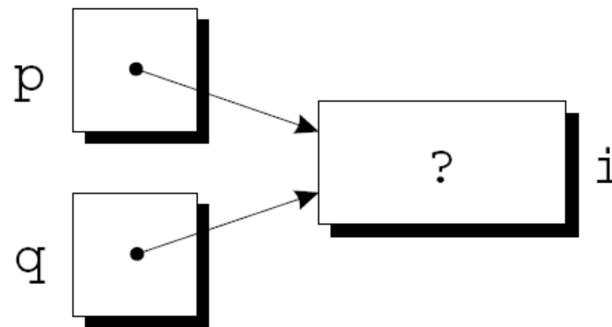
```
p = &i;
```

Assegnamento ai puntatori

- Possiamo anche assegnare:

  `q = p;`

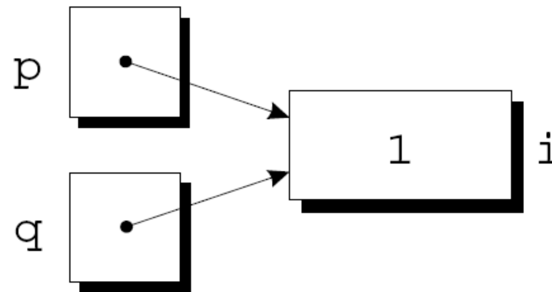
  `q` adesso punta alla stessa variabile puntata da `p`:



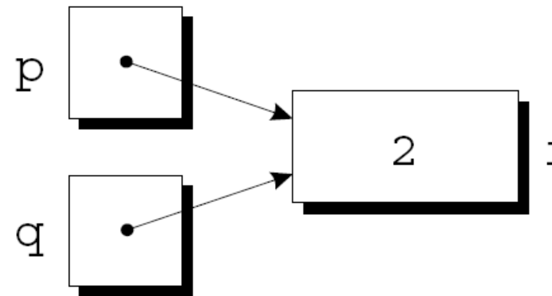
Assegnamento ai puntatori

- Se `p` e `q` puntano entrambi ad `i`, possiamo cambiare `i` assegnando un nuovo valore sia a `*p` che a `*q`:

```
*p = 1;
```




```
*q = 2;
```



- Non c'è limitazione a quanti puntatori possono puntare allo stesso oggetto

Assegnamento ai puntatori

- Attenzione a non confondere

  `q = p;`

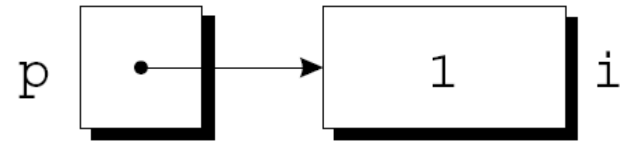
  con

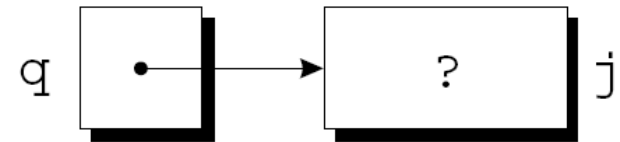
  `*q = *p;`

- La prima istruzione è un assegnamento del puntatore `q` (che punterà allo stesso oggetto puntato da `p`)
- La seconda istruzione copia il valore della variabile puntata da `p` nella variabile puntata da `q`:

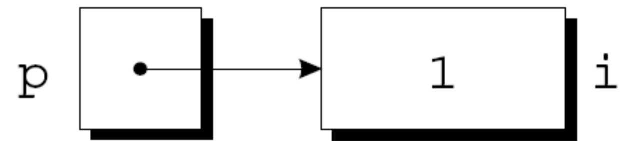
Assegnamento ai puntatori

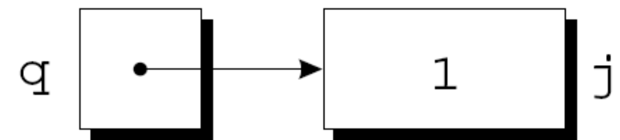
```
p = &i;
q = &j;
i = 1;
```






```
*q = *p;
```





Puntatori come argomenti

- Nel Capitolo 9, abbiamo provato—senza successo—a scrivere la funzione `decompose` per modificare i suoi argomenti
- Non è stato possibile perchè gli argomenti vengono passati **per valore**
- Se però passiamo un *puntatore* alla variabile invece della variabile, allora **sarà il puntatore ad essere passato per valore**. Grazie al puntatore potremo modificare la variabile
- Possiamo quindi scrivere la funzione `decompose` nel modo seguente

Puntatori come argomenti

```
void decompose(double x, long *int_part,
               double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

- Possibili prototipi per `decompose`:

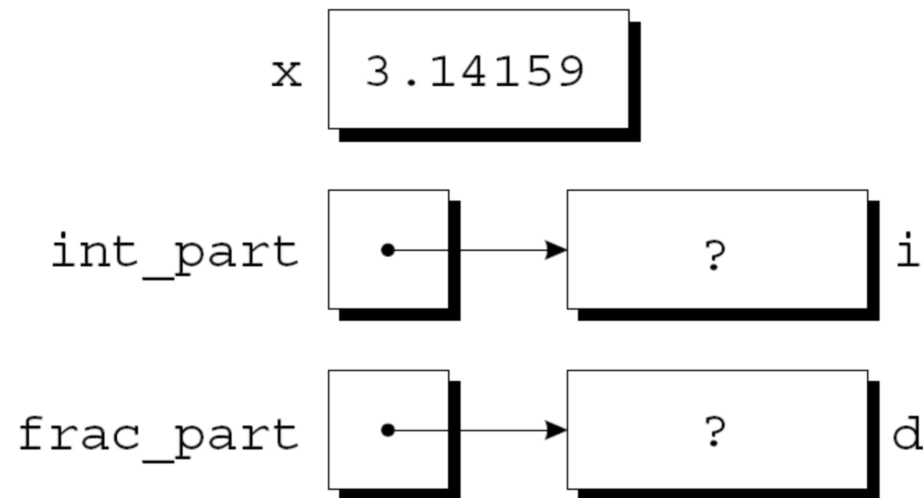
```
void decompose(double x, long *int_part,
               double *frac_part);

void decompose(double, long *, double *);
```

Puntatori come argomenti

- Una chiamata a `decompose`:

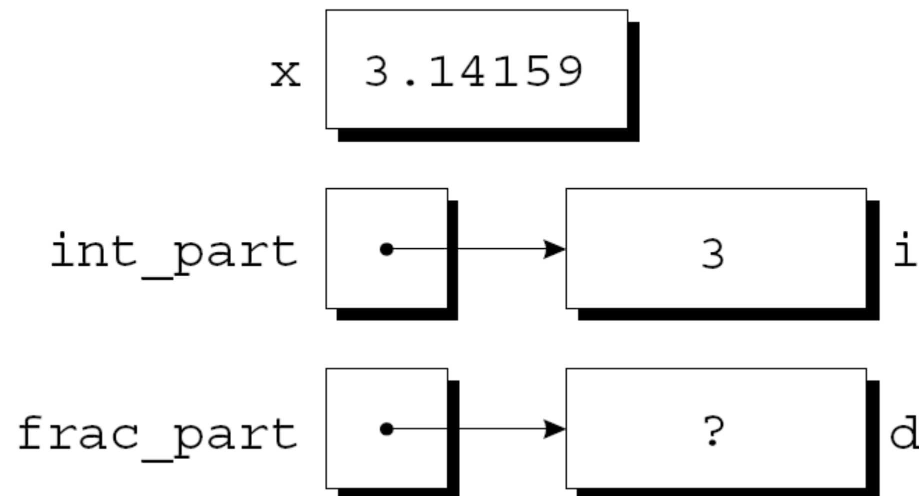
```
decompose(3.14159, &i, &d);
```

- Quando la funzione viene eseguita, `int_part` punta ad `i` e `frac_part` punta a `d`:



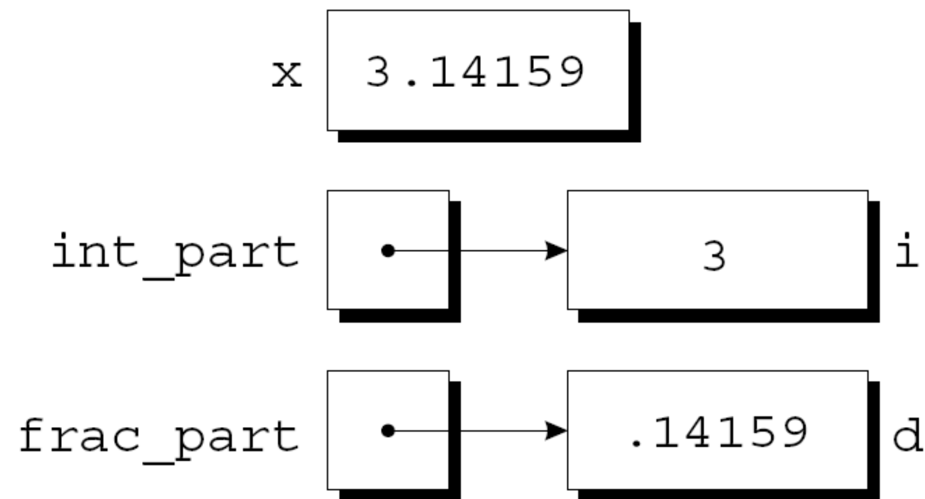
Puntatori come argomenti

- Il primo assegnamento nel corpo di `decompose` converte il valore di `x` al tipo `long` e lo memorizza nell'oggetto puntato da `int_part`:



Puntatori come argomenti

- Il secondo assegnamento memorizza `x - *int_part` nell'oggetto puntato da `frac_part`:



Puntatori come argomenti

- Gli argomenti di una chiamata a `scanf` sono puntatori:

```
int i;
…
scanf("%d", &i);
```

Senza l'operatore `&`, la funzione `scanf` non potrebbe cambiare il valore di `i`

Puntatori come argomenti

- Sebbene gli argomenti di `scanf` debbano essere puntatori, **non sempre** è necessario usare `&`:

```
int i, *p;
…
p = &i;
scanf("%d", p);
```

- In questo caso, poichè `p` è già il puntatore alla variabile, usare `&` sarebbe un errore:

```
scanf("%d", &p);    /*** WRONG ***/
```

Puntatori come argomenti

- Se non si passa un puntatore ad una funzione che si aspetta di riceverlo, si possono avere risultati disastrosi
- Ecco una chiamata a `decompose` in cui `&` non c'è:

```
decompose(3.14159, i, d);
```

- Quando `decompose` assegna dei valori a `*int_part` e `*frac_part`, invece di modificare `i`, cercherà di scrivere in una zona di memoria sconosciua
- Se abbiamo fornito un prototipo della funzione, allora il compilatore rileverà l'errore

Programma: max e min di un array

- Il programma `max_min.c` usa la funzione `max_min` per trovare il più grande ed il più piccolo elemento di un array
- Prototipo per `max_min`:

```
void max_min(int a[], int n, int *max, int *min);
```

- Esempio di chiamata `max_min`:

```
max_min(b, N, &big, &small);
```

- Quando `max_min` trova l'elemento più grande in `b`, memorizza il suo valore in `big` assegnandolo a `*max`.
- `max_min` memorizza il più piccolo elemento di `b` nella variabile `small` assegnandolo a `*min`.

Programma: max e min di un array

- `max_min.c` legge 10 numeri, li memorizza in un array e passa l'array alla funzione `max_min`, e quando la funzione termina stampa il risultato:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);
```

```
  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Usare `const` per proteggere gli argomenti

- Quando un argomento è un puntatore ad una variabile `x`, molto probabilmente il valore di `x` dovrà essere modificato:

  ```
  f(&x);
  ```

- È possibile tuttavia che `f` debba solo esaminare il valore `x`, senza cambiarlo
- Il motivo per l'utilizzo del puntatore potrebbe essere l'**efficienza**: passare una variabile per valore significa copiare alcuni (o molti) byte consumando tempo e memoria

Usare `const` per proteggere gli argomenti

- Possiamo usare `const` per dichiarare esplicitamente che la funzione non cambierà il valore dell'oggetto puntato dall'indirizzo passato alla funzione
- `const` va messo nella dichiarazione della funzione prima del tipo:

```
void f(const int *p)
{
   *p = 0;    /*** WRONG ***/
}
```

Cercare di modificare `*p` è un errore che verrà rilevato dal compilatore

Puntatori come valore di ritorno

- Una funzione può restituire un puntatore:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- Ecco una chiamata alla funzione `max`:

```
int *p, i, j;
…
p = max(&i, &j);
```

Dopo la chiamata, `p` punta o ad `i` oppure a `j`

Puntatori come valore di ritorno

- La funzione `max` restituisce uno dei due puntatori che sono stati forniti come argomenti

- Attenzione a non restituire un puntatore ad una variabile locale:

```
int *f(void)
{
   int i;
   …
   return &i;
}
```

La variabile `i` non esisterà quando `f` termina

Puntatori come valore di ritorno

- Un puntatore può puntare ad elementi di un array
- Se `a` è un array, `&a[i]` è un puntatore all'elemento `i` di `a`
- A volte è utile che una funzione restituisca il puntatore ad un elemento di un array
- Ecco una funzione che restituisce un puntatore all'elemento centrale dell'array `a`, assumendo che `a` abbia `n` elementi:

```
int *find_middle(int a[], int n){
  return &a[n/2];
}
```

Nota: Array come argomenti

- Quando viene passato ad una funzione **il nome di un array viene trattato come un puntatore**
- Esempio:

```
int find_largest(int a[], int n) {
    int i, max;

    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}
```

- Se `b` è un array di `int`, una chiamata a `find_largest` è:

  `largest = find_largest(b, N);`

  Poichè `b` è un array il suo valore è un puntatore all'array, pertanto tale valore sarà copiato nel parametro `a` della funzione. ***L'array non viene copiato!***

… arrivederci alla prossima lezione