# Programmazione (imperativa)

## Corso di Laurea in Informatica

Roberto Cordone

DI - Università degli Studi di Milano

| | | |
|---|---|---|
| Lezioni: | Lunedì 12.00 - 13.00 e 14.00 - 16.00 | Mercoledì 14.00 - 17.00 |
| Laboratorio: | Giovedì 12.00 - 13.00 e 14.00 - 17.00 | |
| Ricevimento: | su appuntamento | |
| Tel.: | 02 503 16235 | |
| E-mail: | roberto.cordone@unimi.it | |
| Web page: | http://homes.di.unimi.it/~cordone/courses/2015-prog/2015-prog.html | |

# Puntatori

I processori organizzano la memoria in $N$ celle

- di dimensione fissata (tipicamente 1 byte = 8 bit)
- ordinate sequenzialmente
- dotate di un indirizzo numerico (da 0 a $N - 1$)

Ogni variabile occupa un numero intero di celle consecutive conoscibile tramite `sizeof` (così pure elementi di vettori e campi di strutture)

Indirizzo della variabile è l'indirizzo della prima cella occupata

Gli indirizzi hanno tipi specifici, non interi, ma convertibili in interi: cambia l'intervallo di definizione, le operazioni, il formato di stampa (esadecimale)

# Tipi puntatori

Variabile puntatore è una variabile che conserva l'indirizzo di un oggetto (variabile, elemento di vettore o campo di struttura)

Per definirne una, occorre specificare

- il tipo dell'oggetto puntato (un tipo qualsiasi)
- il nome della variabile

*tipo_puntato* $*$*variabile* ;

Il tipo della *variabile* si indica con *tipo_puntato* $*$

Nelle dichiarazioni miste l'asterisco va con la variabile:

```
int i, *p, V[10];
```

```
int i;
int *p;
int V[10];
```

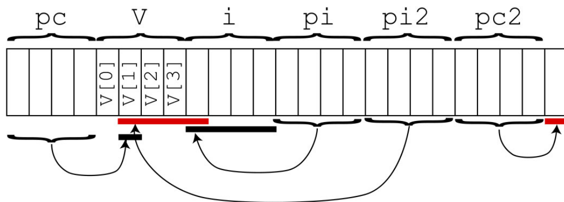
# Osservazioni

Nell'uso corretto, una variabile puntatore indica la prima cella di un oggetto, ma

1. vi si può inserire l'indirizzo d'una cella qualsiasi
   - anche vuota (cioè con informazioni non significative)
   - anche in mezzo a una variabile (elemento o campo)
2. si può inserire lo stesso indirizzo in variabili di tipo diverso

Ovviamente, questo in genere non ha senso e provoca errori

```
char *pc, *pc2;
char V[4];
int i;
int *pi, *pi2;
```



# Referenzazione

L'operatore indirizzo (&) fornisce l'indirizzo di un oggetto

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

L'operatore & ha priorità intermedia fra = e [], fra = e .

```
int V[4];
struct { int x, y } A;
int *p1, *p2;
p1 = &V[3];   equivale a   (p1 = (&(V[3])));
p2 = &A.x;    equivale a   (p2 = (&(A.x)));
```

# Dereferenziazione

L'operatore asterisco (∗) fornisce l'oggetto puntato da un puntatore

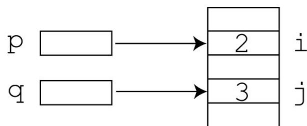
```
int i, j, *p;
p = &i;
j = *p;
```

Se un puntatore non è inizializzato (non contiene l'indirizzo della prima cella di un oggetto del tipo corretto), accedere all'oggetto porta a un comportamento indefinito (valori errati, operazioni casuali, blocco)

Spesso il compilatore lo segnala come avvertimento, non come errore

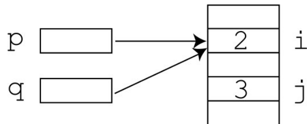
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando `*p` cambia `i` e viceversa (finché non si modifica ancora `p`)

```
int i = 2, j = 3;
int *p = &i, *q = &j;
```



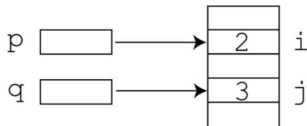

Assegnando un puntatore p a un altro q, essi puntano lo stesso oggetto: modificando *p, cambia *q e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```



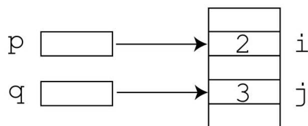

Assegnando l'oggetto *p puntato da un puntatore p all'oggetto *q puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando *p, non cambia *q

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



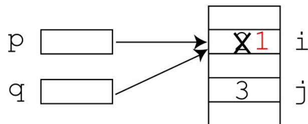
# Assegnamento fra puntatori

```
int i = 2, j = 3;
int *p = &i, *q = &j;
```



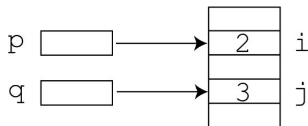

Assegnando un puntatore p a un altro q, essi puntano lo stesso oggetto: modificando *p, cambia *q e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```



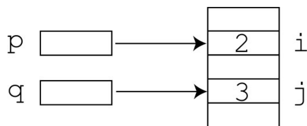

Assegnando l'oggetto *p puntato da un puntatore p all'oggetto *q puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando *p, non cambia *q

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



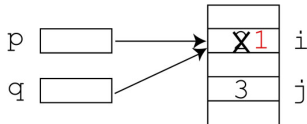
# Assegnamento fra puntatori

```
int i = 2, j = 3;
int *p = &i, *q = &j;
```



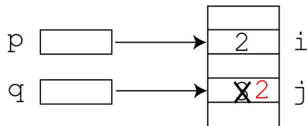

Assegnando un puntatore p a un altro q, essi puntano lo stesso oggetto: modificando *p, cambia *q e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```



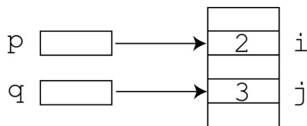

Assegnando l'oggetto *p puntato da un puntatore p all'oggetto *q puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando *p, non cambia *q

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



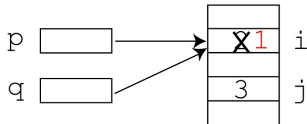
# Assegnamento fra puntatori

```
int i = 2, j = 3;
int *p = &i, *q = &j;
```



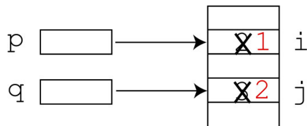

Assegnando un puntatore p a un altro q, essi puntano lo stesso oggetto: modificando *p, cambia *q e viceversa

```
q = p; { ora *q vale 2, j vale 3 }
*p = 1; { ora i e *q valgono 1 }
```




Assegnando l'oggetto *p puntato da un puntatore p all'oggetto *q puntato dall'altro, si hanno due oggetti uguali, ma non lo stesso: modificando *p, non cambia *q

```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



# Aritmetica dei puntatori (1)

Si possono applicare ai puntatori gli operatori di relazione (`==` `!=` `<` `<=` `>` `>=`): prima si convertono gli indirizzi in interi, poi si confrontano

Questo ha senso pratico solo per puntatori a elementi di un vettore

```
&V[1] < &V[3]
```

Si può sommare un intero `j` a un puntatore `p` o sottrarglielo:
si ottiene un indirizzo incrementato o decrementato di `j` volte la dimensione degli oggetti puntati da `p`

`(int) (p + j)` vale `(int) p + sizeof(*p) * j`

Questo ha senso pratico solo per puntatori a elementi di un vettore:
se `p` è l'indirizzo di `V[i]`, `p + j` è l'indirizzo di `V[i+j]`

# Aritmetica dei puntatori (2)

Si possono sottrarre due puntatori di ugual tipo: si ottiene la differenza fra gli indirizzi divisa per la dimensione degli elementi puntati

```
q - p = ((int) q - (int) p) / sizeof(*p)
```

Questo ha senso pratico solo per puntatori a elementi di un vettore, dove si ottiene la differenza fra gli indici degli elementi puntati

se `p` punta `V[i]` e `q` punta `V[j]`, `q - p` vale `j-i`

In pratica, somme e differenze assumono come unità gli elementi puntati

L'aritmetica dei puntatori permetteva di scorrere rapidamente i vettori

```
int V[4], *p;
for (p = &V[0]; p < &V[4]; p++)
{ /* per indicare V[i], si usava l'alias *p */ }
  ...
}
```

Oggi non ci sono vantaggi rispetto a usare gli indici

# Equivalenza vettori-puntatori

L'aritmetica dei puntatori e la struttura ordinata dei vettori implicano che un vettore e il puntatore alla cella di indice 0 sono equivalenti

`V` equivale a `&V[0]`

Si può assegnare un vettore a un puntatore di tipo compatibile

```
int i, V[10];
int *p;
p = V;
```

e da lì in poi usare il puntatore come se fosse un vettore

```
i = p[5];
```

È una conversione implicita e non è lecita la conversione contraria: scrivendo `V = p;` si ottiene un errore

# Vettori e puntatori

- Si può trattare un sottovettore come se fosse un vettore

```
int V[N], *p, s;
s = Somma(V,N);                    somma da V[0] a V[N-1]
p = &V[N/2];
s = Somma(p,N/2);          somma da V[N/2] a V[N/2+N/2-1]
```

- Si può definire un vettore con indice iniziale diverso da 0

```
int V[D-S+1], *p;
p = &V[-S];
```

Il vettore `p` va da `p[S]` a `p[D]`

(`S` *e* `D` *possono anche essere negativi!*)

- Si può definire un vettore dinamico, cioè la cui dimensione è determinata durante l'esecuzione (*vedremo poi come*)