Laboratorio di Algoritmi

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Martedì 8.30 - 10.30 in aula 3	Mercoledì 10.30 - 13.30 in aula 2
	Giovedì 15.30 - 18.30 in aula 2	Venerdì 10.30 - 12.30 in aula 3
Ricevimento:	su appuntamento (Dipartimento di Informatica)	
Tel.:	02 503 16235	
E-mail:	roberto.cordone@unimi.it	
Pagina web:	http://homes.di.unimi.it/~cordone/courses/2020-algo/2020-algo.html	

Puntatori

I processori organizzano la memoria in N celle

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

Ogni variabile (o elemento di vettore, o campo di record)

- occupa un numero intero di celle consecutive
  che si può conoscere con la funzione `sizeof`
- ha come indirizzo quello della prima cella occupata

Tipi puntatore

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

Sono numeri interi, ma hanno un tipo specifico perché

- l'intervallo di definizione non è quello dei numeri interi
- le operazioni eseguibili sono diverse
- il formato di stampa è esadecimale anziché decimale

Una variabile puntatore si definisce indicando tipo e nome, come sempre, e il tipo puntatore si indica con *tipo_puntato* *

```
char *pc;
int *pi;
double *pd;
```

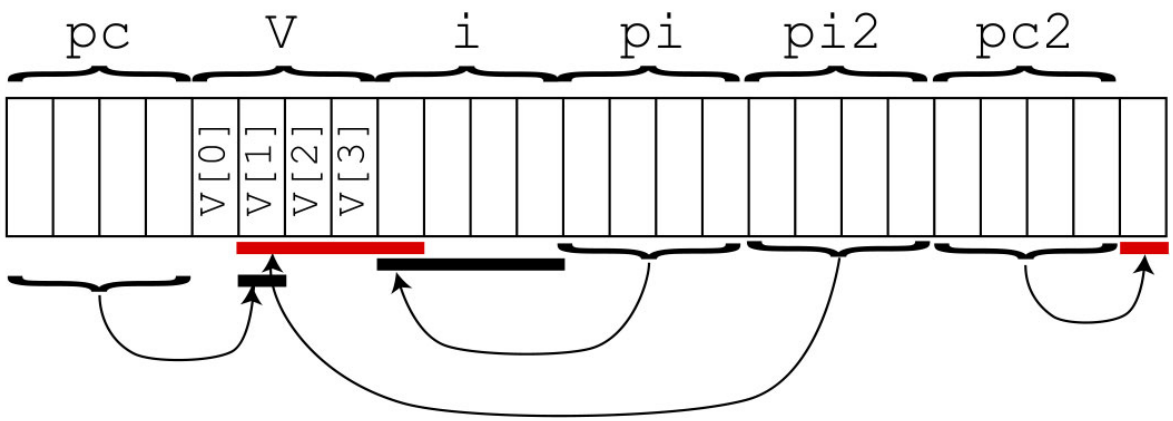
Osservazioni

Una variabile puntatore dovrebbe indicare la prima cella d'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

Questi usi in genere non hanno senso e provocano errori

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



Referenziazione e dereferenziazione

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

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

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

```
j = *p;
```

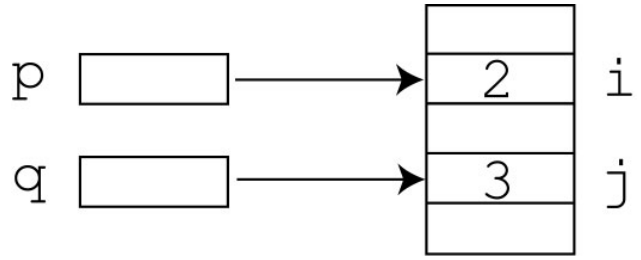
Se un puntatore non è inizializzato, accedere all'oggetto puntato genera comportamenti indefiniti (valori errati, operazioni casuali, blocco)

Spesso il compilatore lo segnala come avvertimento, non come errore

Assegnamento fra puntatori

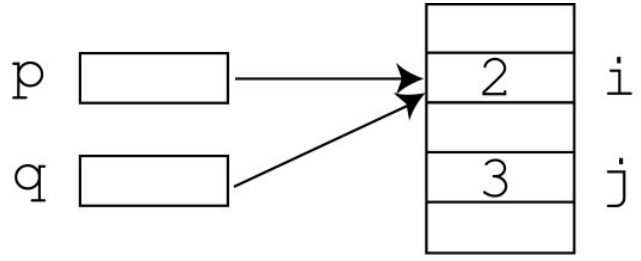
Se p punta i, cioè ne contiene l'indirizzo, *p diventa un alias di i, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

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



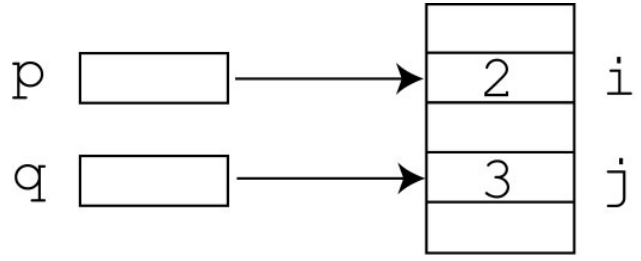
Assegnando un puntatore p a un altro q, essi puntano lo stesso oggetto: modificando *p, cambia *q e viceversa

```
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

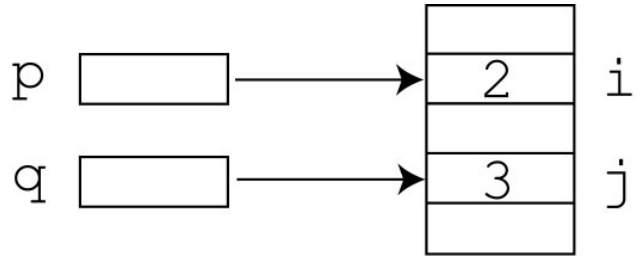
```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



Assegnamento fra puntatori

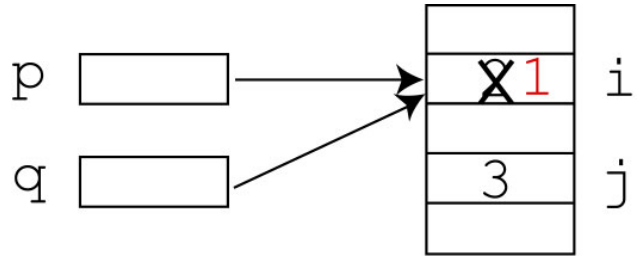
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

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



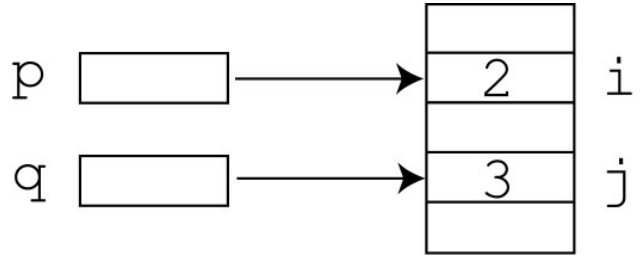
Assegnando un puntatore `p` a un altro `q`, essi puntano lo stesso oggetto: modificando `*p`, cambia `*q` e viceversa

```
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`

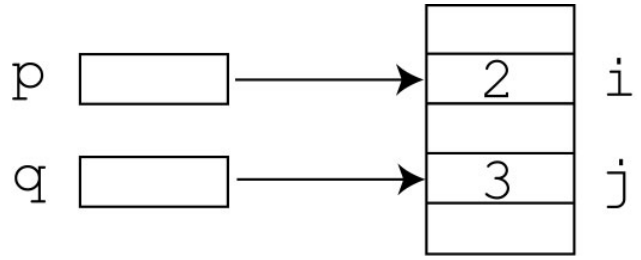
```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



Assegnamento fra puntatori

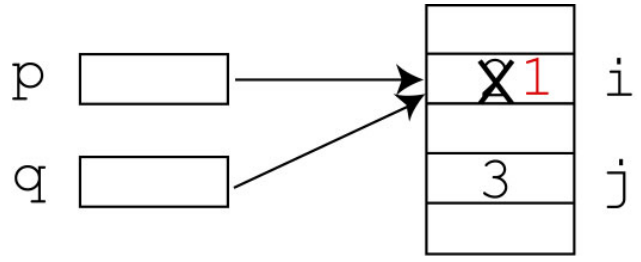
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

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



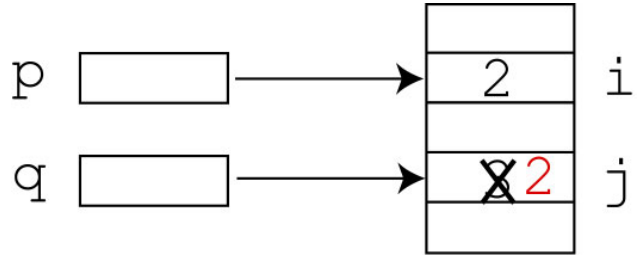
Assegnando un puntatore `p` a un altro `q`, essi puntano lo stesso oggetto: modificando `*p`, cambia `*q` e viceversa

```
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`

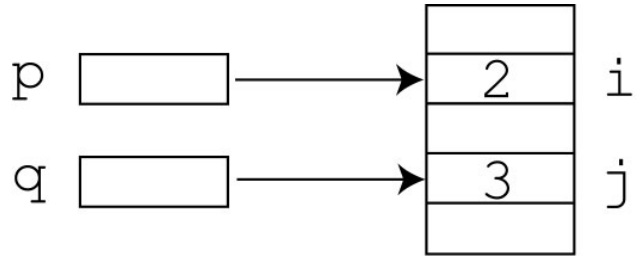
```
*q = *p; { ora j e *q valgono 2 }
*p = 1; { j e *q valgono ancora 2 }
```



Assegnamento fra puntatori

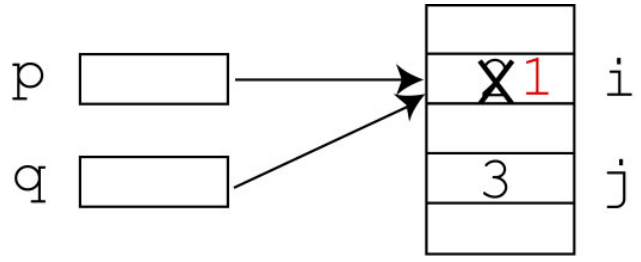
Se `p` punta `i`, cioè ne contiene l'indirizzo, `*p` diventa un alias di `i`, cioè un altro nome dello stesso oggetto: cambiando uno cambia l'altro

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



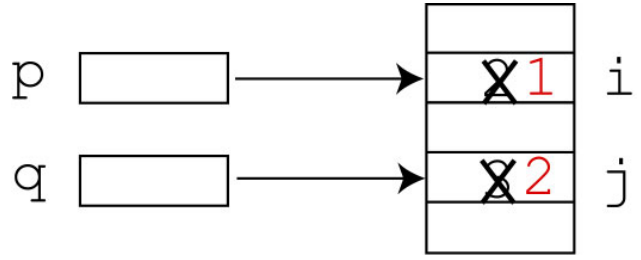
Assegnando un puntatore `p` a un altro `q`, essi puntano lo stesso oggetto: modificando `*p`, cambia `*q` e viceversa

```
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

Siccome i puntatori sono in effetti numeri interi, è lecito

- applicare ai puntatori gli operatori di relazione (`==` `!=` `<` `<=` `>` `>=`)
- sommare qualsiasi intero `j` a un puntatore `p`: si ottiene un puntatore all'indirizzo `p` spostato di `j` volte la dimensione degli oggetti puntati

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

  Non è una banale somma di interi!

Questa operazione ha senso 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]`

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*)

Allocazione dinamica della memoria

A una variabile puntatore si può assegnare un'area di memoria allocata dinamicamente, cioè un intervallo di celle consecutive nuovo prese in un'area della memoria detta *heap*

Questo consente di gestire strutture dati più flessibili di quelle statiche

- strutture di dimensione costante, ma incognita prima dell'esecuzione oppure variabile, ma con un massimo determinato al principio
- strutture di dimensione completamente libera durante l'esecuzione

La funzione `malloc`, dichiarata nella libreria `stdlib.h`

- richiede il numero totale di celle assegnate che può essere variabile e che conviene calcolare con `sizeof`
- rende un puntatore da convertire nel tipo giusto

`p = (`*tipo* `*) malloc(`*dimensione*`);`

Esempio: allochiamo spazio per un numero intero

```
int *pi;
pi = (int *) malloc(sizeof(int));
```

Vettori dinamici

I vettori dinamici sono sequenze di oggetti omogenei il cui numero è fissato al momento dell'allocazione, ma varia da un'esecuzione all'altra

Si dichiarano con la funzione `calloc`, che alloca il blocco e lo azzera

```
int *p = (int *) calloc(n+1,sizeof(int));
```

dove `n` è una variabile intera

Il blocco di memoria è trattabile come un vettore statico:
si accede agli elementi in lettura e scrittura con gli indici numerici

```
i = p[4];
p[5] = 7;
```

Quando l'allocazione fallisce, viene restituito il puntatore nullo (`NULL`)
prima di usare l'area occorre verificare che il puntatore non sia nullo

```
if (p == NULL) printf("Errore di allocazione!\n");
```

Deallocazione

L'area allocata è marcata così che successive allocazioni non la usino

Quando non occorre più, l'area allocata va deallocata (resa disponibile) per evitare il memory leakage (consumo progressivo della memoria)

`free(`*puntatore*`);`

dove *puntatore* è l'indirizzo della prima cella dell'area

- un'area va deallocata una volta sola, anche se ha più puntatori
- si deallocano solo puntatori a oggetti dinamici, non statici
- dopo la deallocazione, conviene fissare il puntatore a `NULL`, altrimenti continua a indicare l'area, ora libera
- il puntatore deve indicare la prima cella dell'area da liberare (potrebbe essere stato scalato per ottenere vettori da `S` a `D`)

Matrici dinamiche

Matrice dinamica è un vettore dinamico di puntatori a vettori dinamici

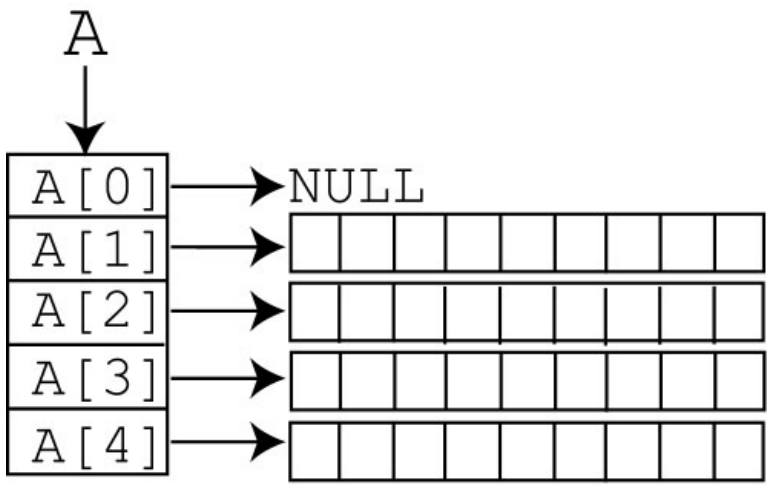
Serve a ottenere matrici di dimensione incognita a priori
ed eventualmente matrici con righe non uniformi

Allocazione

```
int **A;
int m, n, i, j;
A = (int **) calloc(m+1,sizeof(int *));
for (i = 1; i <= m; i++)
  A[i] = (int *) calloc(n+1,sizeof(int));
```

Deallocazione

```
for (i = 1; i <= m; i++)
  free(A[i]);
free(A);
```



Funzioni

Funzioni o procedure sono brani di codice abbastanza importanti da avere un nome, dei dati e dei risultati, come i programmi

L'uso delle funzioni consente di

- dividere un programma in brani di codice
  - più facili da capire
  - più facili da modificare
- non ripetere brani di codice identici o molto simili
- riutilizzare brani di codice in programmi diversi

Esempio

Calcolo dei quadrati e dei cubi per i primi dieci numeri positivi

```c
#include <stdio.h>
#include <stdlib.h>

int quadrato (int y);
int cubo (int y);

int main (int argc, char *argv[])
{
  int x;
  int Q[11];
  int C[11];

  for (x = 1; x <= 10; x++)
    Q[x] = quadrato(x));

  for (x = 1; x <= 10; x++)
    C[x] = cubo(x));

  return EXIT_SUCCESS;
}
```

```c
int quadrato (int y)
{
  return y * y;
}

int cubo (int y)
{
  int q;

  q = quadrato(y);
  return q * y;
}
```

Prototipo, definizione e chiamate

Il prototipo (dichiarazione) specifica

- il tipo del risultato della funzione (`void` se non dà risultati)
- il nome della funzione
- i dati richiesti dalla funzione

La definizione della funzione riprende il prototipo e specifica le variabili locali e le operazioni eseguite dalla funzione

Chiamata di una funzione (invocazione) è
ciascun uso della funzione in espressioni del `main` o di altre funzioni e specifica i dati sui quali effettivamente la funzione opera nel caso specifico

Prototipo

tipo funzione(*tipo1 nome1*, *tipo2 nome2*, ...);

Il prototipo specifica

- il *tipo* del risultato della funzione (`void` se non dà risultati)
- il *nome* della funzione
- il *tipo* e il *nome* di ciascun dato

I parametri formali sono i nomi dei dati indicati nel prototipo
I parametri attuali saranno i valori dei dati indicati nelle chiamate

Il prototipo mette il nome della funzione nella tabella dei simboli
(essendo un'istruzione, termina con ;)

Definizione

```
tipo funzione(tipo1 nome1,tipo2 nome2,...)
{
   parte dichiarativa
   parte esecutiva
   [return espressione;]
}
```

- l'intestazione riprende il prototipo senza il ; (*non è un'istruzione*)
- il corpo fra parentesi graffe specifica variabili locali e istruzioni e termina con l'istruzione `return` seguita da un'*espressione*:
  risultato della funzione è il valore dell'espressione

Una procedura non restituisce risultati: è di tipo `void` e
non occorre che termini con `return`

Chiamata

Ogni chiamata di una funzione (invocazione) è
il suo uso in espressioni del `main` o di altre funzioni

oggetto = *funzione*(*parametri attuali*);
procedura(*parametri attuali*);

Parametri attuali sono le espressioni i cui valori forniscono i dati

Si chiamano anche argomenti

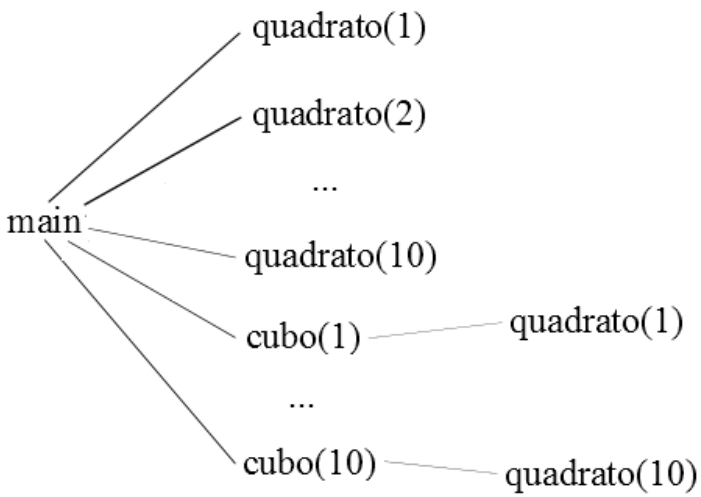
Per ogni funzione si possono avere molte chiamate su dati diversi

- da parte della stessa funzione
- da parte di diverse funzioni

Struttura modulare

Le chiamate a funzione sono organizzate ad albero, radicato nel `main`

- la funzione chiamante indica quali funzioni chiamate devono operare e in che ordine e fornisce a ciascuna i dati su cui operare
- le funzioni chiamate restituiscono alla funzione chiamata i risultati del loro lavoro
- ogni funzione chiamata può avere sottoprocedure, con le quali organizza il lavoro allo stesso modo (da cui la struttura ad albero)
- la funzione chiamante conosce solo dati e risultati; non sa quali operazioni compiono e che mezzi usano le funzioni chiamate



Gestione dello *stack*

Il processore riserva alle funzioni un'area di memoria detta *stack* che viene gestita come una pila e che è completamente separata dallo *heap*

- alla chiamata, alloca in cima alla pila lo spazio necessario a
  - parametri formali (dati)
  - variabili locali (risultati parziali)
  - risultato della funzione
- al termine della funzione, dealloca lo spazio e rende il risultato

A causa della deallocazione, parametri e variabili locali esistono solo fra la chiamata e il termine della funzione

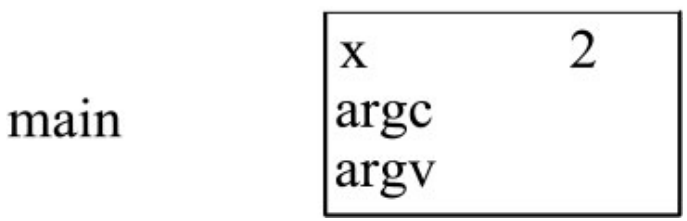
Le strutture dinamiche stanno in un'altra area di memoria detta *heap*:
vengono deallocate solo da `free` o al termine dell'intero programma

Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: il `main` esegue l'istruzione `cubo(x);`



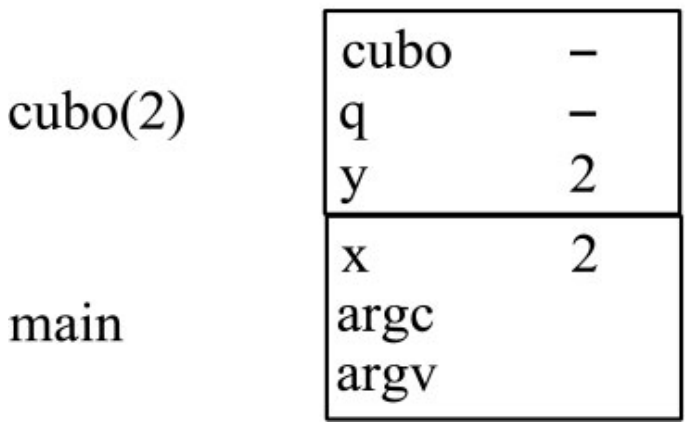
Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: il `main` esegue l'istruzione `cubo(x);`

1. il parametro attuale è `x` e vale 2
2. si alloca sullo *stack* spazio per `y`, `q` e il risultato di `cubo`
3. si assegna il valore 2 a `y`

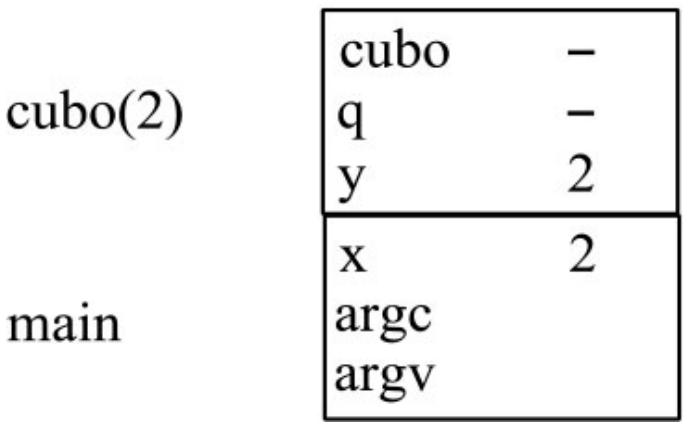


Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: la funzione `cubo` esegue l'istruzione `quadrato(y);`



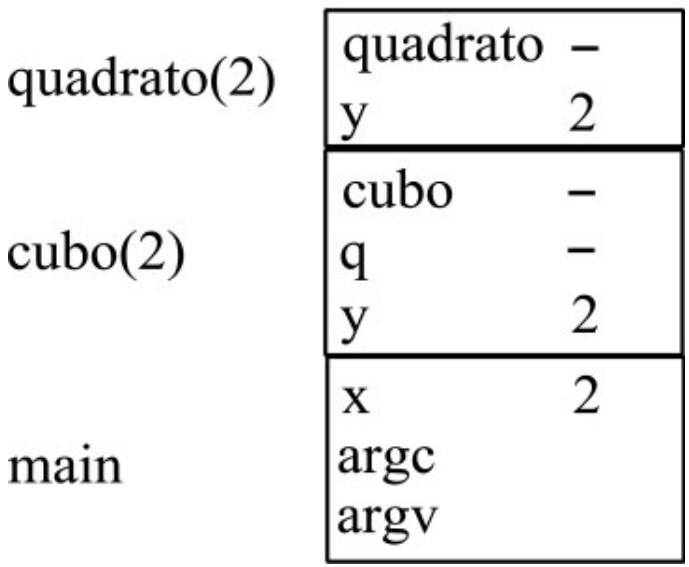
Gestione dello *stack*: esempio

Alla chiamata della funzione, il processore:

1. valuta i parametri attuali
2. alloca in cima allo *stack* lo spazio necessario
3. assegna i valori dei parametri attuali ai parametri formali

Esempio: la funzione `cubo` esegue l'istruzione `quadrato(y);`

1. il parametro attuale è `y` e vale 2
2. si alloca sullo *stack* spazio per `y` e il risultato di `quadrato`
3. si assegna il valore 2 a `y`

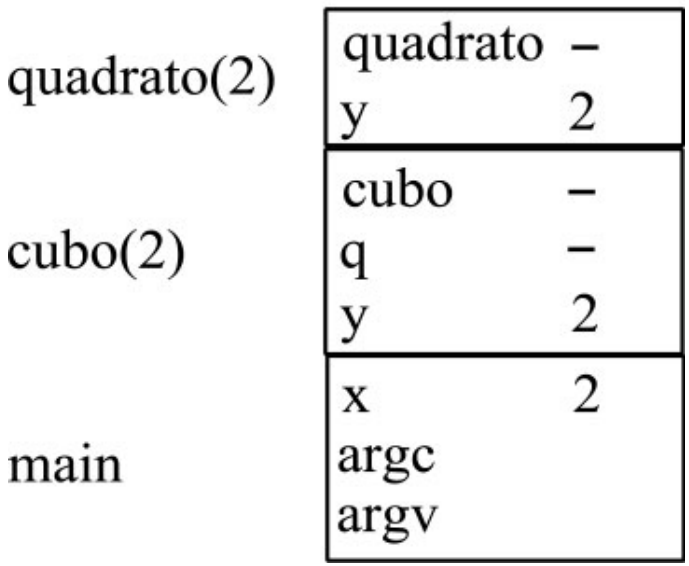


Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `quadrato` esegue l'istruzione `return y * y;`



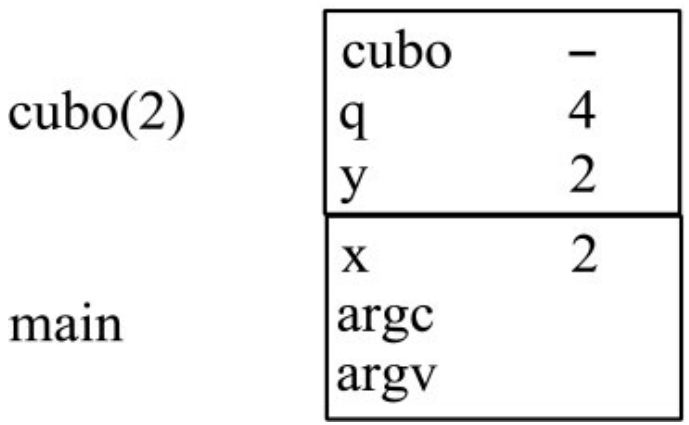
Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `quadrato` esegue l'istruzione `return y * y;`

1. il risultato `y * y` vale 4
2. si dealloca lo spazio per `y` e per il risultato (salvando questo a parte)
3. si restituisce 4 alla funzione `cubo`, che lo assegna a `q`

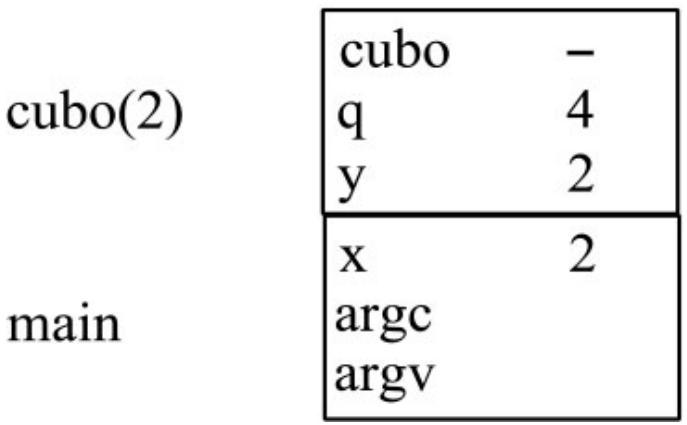


Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `cubo` esegue l'istruzione `return q * y;`



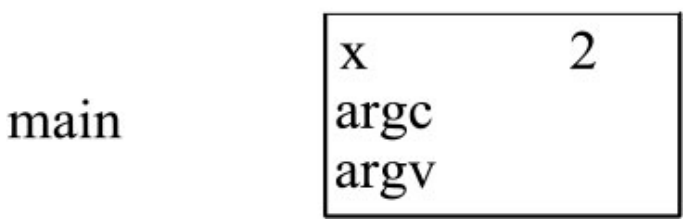
Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione che segue l'istruzione `return` come risultato
2. dealloca dalla cima allo *stack* lo spazio dell'ultima chiamata
3. restituisce il risultato alla funzione chiamante

Esempio: la funzione `cubo` esegue l'istruzione `return q * y;`

1. il risultato `q * y` vale 8
2. si dealloca lo spazio per `y`, `q` e per il risultato (salvando questo a parte)
3. si restituisce 8 al `main`, che lo assegna a `C[x]`



Passaggio dei parametri

Ad ogni chiamata i parametri attuali vengono valutati e sostituiscono i parametri formali nella funzione chiamata

In generale vi sono due modi di passare i parametri

- per valore: la funzione accede a copie dei dati e le eventuali modifiche interne vanno perse
- per indirizzo: la funzione accede ai dati originali e le eventuali modifiche interne si conservano

Passaggio per valore

Il passaggio dei parametri per valore non ha effetti collaterali

Si possono usare i parametri formali come variabili locali, modificandoli per produrre il risultato senza sporcare i parametri attuali

```
int main (int argc, char *argv[])
{
  int x, q;

  x = 10;
  q = quadrato(x);
  /* x vale ancora 10 */
  return EXIT_SUCCESS;
}
```

```
int quadrato (int y)
{
  y = y * y;
  /* ora y vale 100 */
  return y;
}
```

Passaggio per indirizzo

Il passaggio di parametri per indirizzo consente effetti collaterali

Quindi consente alla funzione di accedere ai dati originali e modificarli

Ma in C il passaggio per indirizzo non esiste!

```
void scambia (int a, int b);
int main (int argc, char argv[])
{
  int a, b;
  a = 1;
  b = 2;
  scambia(a,b);
  /* ora a vale ancora 1
       e b vale ancora 2 */
  return EXIT_SUCCESS;
}
```

```
void scambia (int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
  /* ora temp vale 1,
     a vale 2 e b vale 1 */
}
```

Passaggio per indirizzo in C

Per simulare il passaggio per indirizzo in C si usa un trucco:

si passa per valore l'indirizzo del parametro

```
void scambia (int *pa, int *pb);

int main (int argc, char argv[])
{
  int a, b;
  a = 1;
  b = 2;
  scambia(&a,&b);
  /* ora a vale 2 e b vale 1 */
  return EXIT_SUCCESS;
}
```

```
void scambia (int *pa, int *pb)
{
  int temp;
  temp = *pa;
  *pa = *pb;
  *pb = temp;
  /* ora temp vale 1,
     a vale 2 e b vale 1 */
}
```

Risultati multipli o vettoriali

Un altro limite del passaggio per valore è che

- si può restituire come risultato un solo valore
- il risultato non può essere un vettore statico
  (dato che non si può usare un vettore statico in un assegnamento)

Ma se occorre restituire più valori o un vettore di valori?

- risultati multipli:
  - si restituisce un record contenente i valori come campi
  - si passano dei finti "dati" che vengono modificati sfruttando gli effetti collaterali del passaggio per indirizzo
- vettore:
  - si restituisce un "vettore" dinamico (puntatore);
    l'area di memoria va allocata nella funzione e deallocata fuori
  - si passa l'intero vettore come dato fittizio, ma per indirizzo (dunque non il vettore, ma un puntatore al vettore)

Risultati multipli

Decomposizione di un numero double nella parte intera e frazionaria

```
void decompone (double d, long *pl, double *pf);
int main (int argc, char argv[])
{
  double d, f;
  long l;

  d = 3.5;
  decompone(d,&l,&f);
  /* ora l vale 3L e f vale 0.5 */
  return EXIT_SUCCESS;
}
```

```
void decompone (double d, long *pl, double *pf)
{
  *pl = (long) d;
  *pf = d - *pl;
}
```

Dati vettoriali

Alla chiamata, i vettori vengono convertiti in puntatori e passati

- Non si deve specificarne la lunghezza, anche se sono statici

  Non `int funzione(int V[LUNGHEZZA]);`

  ma `int funzione(int *V);`

  Se occorre conoscerla, la lunghezza si passa come parametro
- gli elementi di un vettore sono passati per indirizzo, il vettore per valore: corrispondentemente, le modifiche agli elementi si conservano, quelle al vettore no dato che il puntatore non cambia

Risultati vettoriali

Vi sono due modi per far scrivere una funzione che restituisce un vettore

1. la funzione alloca un vettore dinamico e restituisce il puntatore alla funzione chiamante
2. la funzione chiamante passa per indirizzo un puntatore alla funzione, la funzione alloca un vettore dinamico e modifica il puntatore assegnandogli il vettore dinamico

```
int main (int argc, char argv[])
{
  int *V;
  int n;

  n = 10;
  V = creavettore(n);
  /* ora V e' un vettore dinamico */
  free(V);
  return EXIT_SUCCESS;
}

int *creavettore (int n)
{
  int *V;
  V = (int *) calloc(n+1,sizeof(int));
  return V;
}
```

```
int main (int argc, char argv[])
{
  int *V;
  int n;

  n = 10;
  creavettore(&V,n);
  /* ora V e' un vettore dinamico */
  free(V);
  return EXIT_SUCCESS;
}

void creavettore (int **pV, int n);
{
  *pV = (int *) calloc(n+1,sizeof(int));
}
```

Passaggio delle strutture

Le strutture si passano per valore copiando campo per campo, ma

- se contengono campi puntatore, ci si limita a copiare gli indirizzi: gli oggetti puntati risultano passati per indirizzo

Si usa passare per indirizzo le strutture di grandi dimensioni, anche quando non si vuole modificarle, per rendere più efficiente il passaggio

Si copia un semplice indirizzo anziché molte celle di dati