.

Funzioni

Stefano Ferrari

Università degli Studi di Milano
stefano.ferrari@unimi.it

Programmazione

anno accademico 2016–2017

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

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

```
#include <stdio.h>
#include <stdlib.h>
#include "iocrema.h"

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

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

  for (x = 1; x <= 10; x++)
    StampaIntero(quadrato(x));
  StampaStringa("\n");

  for (x = 1; x <= 10; x++)
    StampaIntero(cubo(x));
  return EXIT_SUCCESS;
}
```

```
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

La chiamata di una funzione (invocazione):

- è l'uso della funzione in espressioni del `main` o di altre funzioni
- 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
  - 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

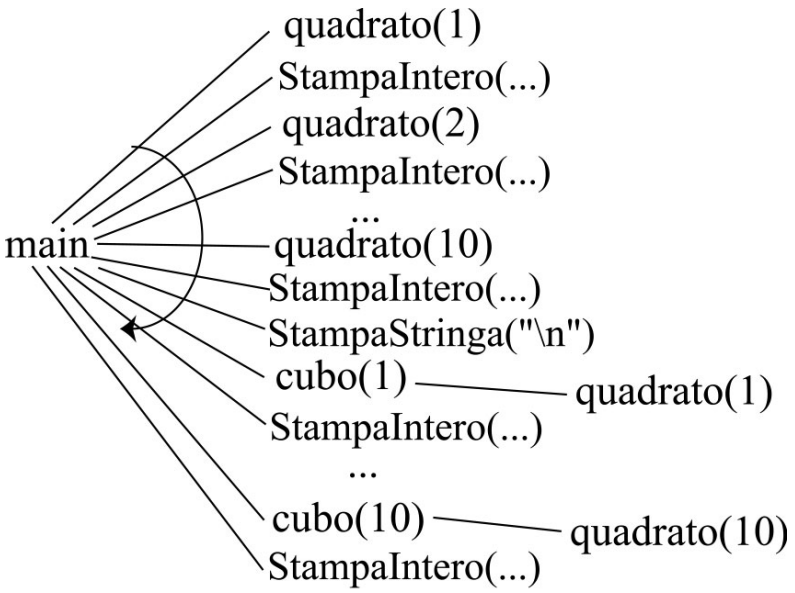
Le chiamate a funzione sono organizzate ad albero

La funzione chiamante e le chiamate in essa contenute
si comportano come capufficio e impiegati

- il capufficio indica quali impiegati devono lavorare e in che ordine e fornisce a ciascuno i dati su cui operare
- gli impiegati forniscono al capufficio i risultati del loro lavoro
- ogni impiegato può avere sottoposti, con i quali organizza il lavoro allo stesso modo (da cui la struttura ad albero)
- il capufficio conosce solo dati e risultati; non sa quali operazioni compiono e che mezzi usano gli impiegati

La radice dell'albero è il `main`

Albero delle chiamate



Gestione dello *stack*

Il processore riserva alle funzioni un'area di memoria detta *stack* che viene gestita come una pila

- 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

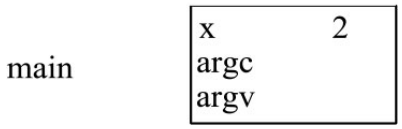
.

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);`



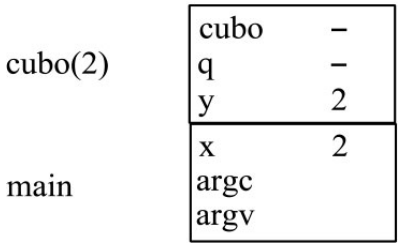
Gestione dello *stack*: esempio

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`



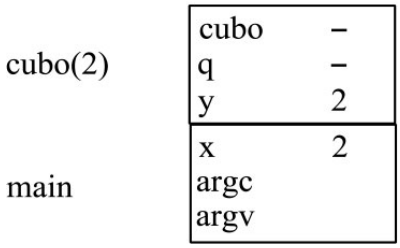
.

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);`



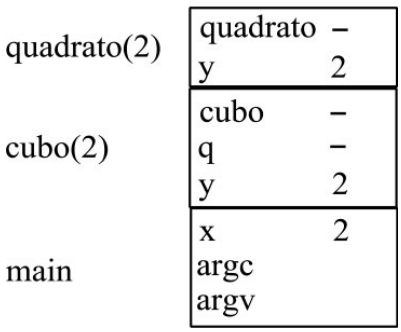
Gestione dello *stack*: esempio

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`



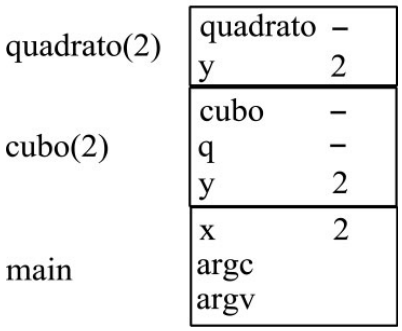
.

Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione dell'istruzione `return` (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;`



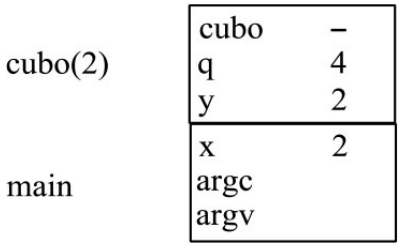
Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione dell'istruzione `return` (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 (salvandolo a parte)
3. si restituisce 4 alla funzione `cubo`, che lo assegna a `q`



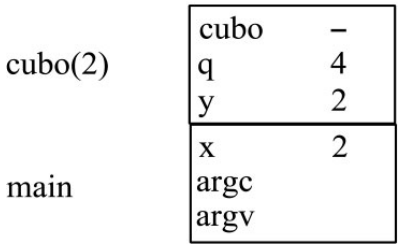
.

Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione dell'istruzione `return` (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;`



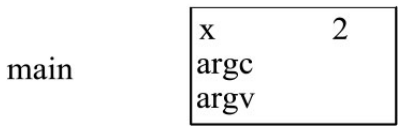
Gestione dello *stack*: esempio

Al termine della funzione, il processore:

1. valuta l'espressione dell'istruzione `return` (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 (salvandolo a parte)
3. si restituisce 8 al `main`, che lo passa alla funzione `StampaIntero`



.

Conversioni di tipo (1)

Il passaggio dei dati e il recupero del risultato obbediscono alle regole implicite per le conversioni di tipo

```
int main (int argc, char *argv[])
{
  int i;
  double d = 3.5;
  i = quadrato(d);
  /* i vale 9, cioe' 3*3 */
  return EXIT_SUCCESS;
}
```

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

Conversioni di tipo (2)

```
int main (int argc, char *argv[])
{
  int i;
  double d = 3.5;
  i = quadrato(d);
  /* i vale 12, cioe' (int) 3.5*3.5 */
  return EXIT_SUCCESS;
}
```

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

```
int main (int argc, char *argv[])
{
  double i;
  double d = 3.5;
  i = quadrato(d);
  /* i vale 12.25, cioe' 3.5*3.5 */
  return EXIT_SUCCESS;
}
```

```
double quadrato (double y)
{
  return y * y;
}
```

.

Scope

Come si è visto nell'esempio, ogni funzione ha un suo spazio riservato sullo *stack* e può agire solo su parametri e variabili locali

- non accede alle variabili delle funzioni chiamate
- non accede alle variabili della funzione chiamante

Si possono introdurre variabili globali dichiarate fuori dal main e visibili in ogni funzione del file

Le variabili globali sono inutili e sconsigliabili

- nascondono il fatto che una funzione le usa e le modifica (bachi)
- rendono complicato chiamare la funzione da funzioni in altri file

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 realizzare 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 (1)

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 una `struct` oppure si passano dei finti "dati" che vengono modificati sfruttando gli effetti collaterali del passaggio per indirizzo
- vettore: si restituisce un "vettore" dinamico (puntatore) con area di memoria allocata nella funzione (e da deallocare fuori) oppure si sfrutta l'equivalenza con i puntatori come indicato in seguito

.

Risultati multipli (2)

Scomposizione di un numero `double` nella parte intera e frazionaria

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

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

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

Passaggio dei vettori

Alla chiamata, i vettori vengono convertiti in puntatori prima di passarli:

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

  `int funzione(int *V);` OK
  `int funzione(int V[LUNGHEZZA]);` NO

  - anche `int funzione(int V[]);` viene accettata, ma comunque `V` viene trattato come un puntatore

  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 scrivere una funzione che ritorni un vettore

1. la funzione alloca un vettore dinamico e restituisce il puntatore alla funzione chiamante

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

  n = 10;
  V = creavettore(n);
  /* 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;
}
```

Risultati vettoriali (2)

Vi sono due modi per scrivere una funzione che ritorni un vettore

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, n;

  n = 10;
  creavettore(&V,n);
  /* 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