Corso “Programmazione 1”
Capitolo 05: Le Funzioni

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Giovanni De Toni - giovanni.detoni@unitn.it
Stefano Berlato - stefano.berlato-1@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 11 ottobre 2021

Terms of Use and Copyright

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Passaggio di parametri

In C++ esistono tre modalità passaggio di parametri a una funzione:

- per valore
- per riferimento
- per puntatore

(Spesso le ultime due sono confuse in letteratura, perché hanno finalità simili.)

Il passaggio di parametri per valore

- Definizione di parametri formali analoga a definizione di variabili
  - Sintassi lista dei parametri: `( tipo identificatore, ...)`
  - Es: `int fact(int n) {...}`
- Simile a definire una nuova variabile locale e assegnarle il valore dell'espressione del parametro attuale.
  - Es: `fact(3*x); //simile a: int n = 3*x;`
- Il parametro formale acquisisce il valore del parametro attuale
  - il parametro attuale può essere un'espressione senza indirizzo
  - può essere di tipo diverso compatibile $\Longrightarrow$ conversione implicita
- L'informazione viene (temporaneamente) duplicata
  $\Longrightarrow$ possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, il parametro attuale non viene modificato
  $\Longrightarrow$ passaggio di informazione solo dalla chiamante alla chiamata
- Tutti gli esempi di funzioni visti finora usano passaggio per valore:
  { `FUNCTIONS/...` }

Il passaggio di parametri per riferimento

- Definizione di parametri formali simile a definizione di riferimenti
  - Sintassi lista dei parametri: `( tipo & identificatore,...)`
  - Es: `int swap(int & n, int & m) {...}`
- Simile a definire un riferimento "locale" ad un'espressione dotata di indirizzo
  - Es: `swap(x,y); //simile a: int & n=x; int & m=y`
- Il parametro è un riferimento al parametro attuale
  - il parametro attuale deve essere un'espressione dotata di indirizzo
  - deve essere dello stesso tipo
- L'informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, modifico il parametro attuale
  $\Longrightarrow$ passaggio di informazione anche dalla chiamata alla chiamante

Esempi

- passaggio per valore, errato:
  { `FUNCTIONS/scambia_err.cc` }
- passaggio per riferimento, corretto [D]:
  { `FUNCTIONS/scambia.cc` }
- passaggio per riferimento non ammesso, tipo diverso:
  { `FUNCTIONS/riferimento_err.cc` }
- problemi ad usare il riferimento quando non dovuto:
  { `FUNCTIONS/mcd_err.cc` }
- restituzione di due valori :
  { `FUNCTIONS/rectpolar.cc` }
- parametro come input e output di una funzione:
  { `FUNCTIONS/iva.cc` }

Passaggio di parametri per riferimento costante

- È possibile definire passaggi per riferimento in sola lettura (passaggio per riferimento costante)
  - Sintassi: `(const tipo & identificatore, ...)`
  - Es: `int fact(const int & n,...) {...}`
- Riferimento: l'informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Non permette di modificare `n`!
  - Es: `n = 5; //ERRORE!`
  - $\Longrightarrow$ passaggio di informazione solo dalla chiamante alla chiamata
  - $\Longrightarrow$ solo un input alla funzione
- Usato per passare in input alla funzione oggetti "grossi"
  - efficiente (no spreco di CPU e memoria)
  - evita errori
  - permette di individuare facilmente gli input della funzione
- Uso di riferimenti costanti:
  { `FUNCTIONS/usa_const.cc` }

Esempi (2)

- esempi per riferimento:
  { `FUNCTIONS/cipeciop.cc` }
- ...:
  { `FUNCTIONS/pippo.cc` }
- ...:
  { `FUNCTIONS/paperino.cc` }
- ...:
  { `FUNCTIONS/topolino.cc` }

Morale

Con i riferimenti è facile fare confusione!

Il passaggio di parametri per puntatore

- Definizione di parametri formali: puntatori passati per valore
  - Sintassi lista dei parametri: `( tipo * identificatore,...)`
  - Es: `int swap(int * pn, int * pm) {...}`
  - N.B.: nella chiamata, si passa l'indirizzo dell'oggetto passato
- Simile a definire un puntatore "locale" ad un'espressione dotata di indirizzo
  - Es: `swap(&x,&y);//simile a: int *pn=&x; int *pm=&y`
- Il parametro è un puntatore al(l'oggetto il cui indirizzo è dato dal) parametro attuale
  - che deve essere un'espressione dotata di indirizzo
  - che deve essere dello stesso tipo
- L'informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, modifico il parametro attuale
  $\Longrightarrow$ passaggio di informazione anche dalla chiamata alla chiamante

$\Longrightarrow$ effetto simile al passaggio per riferimento (vedi C)

Esempi

- come scambia.cc, con passaggio per puntatore:
  { `FUNCTIONS/scambia_punt.cc` }
- come iva.cc, con passaggio per puntatore:
  { `FUNCTIONS/iva2.cc` }
- come paperino.cc, con passaggio per puntatore:
  { `FUNCTIONS/paperino2.cc` }

Passaggio per valore vs. p. per riferimento/puntatore

- Vantaggi del passaggio per riferimento/puntatore:
  - Minore carico di calcolo e di memoria
    (soprattutto con parametri di grosse dimensioni)
  - Permette di restituire informazione da chiamata a chiamante
- Svantaggi del passaggio per riferimento/puntatore:
  - Rischio di confusione nel codice (non si sa dove cambiano i valori)
  - Aliasing (entità con più di un nome)
  - Parametro formale e attuale esattamente dello stesso tipo
  - Si possono passare solo espressioni dotate di indirizzo

Nota

- alcuni linguaggi (es C) non ammettono passaggio per riferimento (solo per puntatore)

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`

Funzioni che restituiscono un riferimento

- Restituisce un riferimento ad un'espressione (con indirizzo)
  - l'espressione deve riferirsi ad un oggetto del chiamante (es. un parametro formale passato per riferimento, un elemento di un array)
  - deve essere dello stesso tipo
- La chiamata è un'espressione dotata di indirizzo!

```
int& max(int& x,int& y) //restituisce un riferimento
{return (x > y ? x : y);}//x,y riferimenti a oggetti
                         //non locali
(...)
int m=44, n=22;
max(m,n) = 55;     //cambia il valore di m da 44 a 55
```

Esempio di cui sopra esteso:
{ FUNCTIONS/restituzione_riferimento.cc }

Sovrapposizione di parametri (overloading)

- In C++ è possibile dare lo stesso nome a funzioni diverse, purché con liste di parametri diverse, per numero e/o per tipo
- Il compilatore “riconosce” la giusta funzione per ogni chiamata.
- In caso di ambiguità, il compilatore produce un errore
- Conversioni implicite ammissibili, purché non causino ambiguità

```
int max(int, int) {...};
int max(int, int, int) {...};
double max(double, double) {...};
(...)
  cout << max(99,77) << "␣" << max(55,66,33) << "␣"
       << max(3.4,7.2) << endl;
  cout << max(3,3.1) << endl; // errore: AMBIGUA
```

Esempio di cui sopra esteso:

{ FUNCTIONS/overloading.cc }

Funzioni con argomenti di default (cenni)

- In C++ è possibile fornire parametri opzionali, con valori di default
  - permette chiamate con liste di parametri attuali ridotte
  - i parametri opzionali devono essere gli ultimi della lista
  - il match viene effettuato da sinistra a destra

```
double p(double, double, double =0, double =0, double =0);

  cout << p(x, 7) << endl;
  cout << p(x, 7, 6) << endl;
  cout << p(x, 7, 6, 5) << endl;
  cout << p(x, 7, 6, 5, 4) << endl;

double p(double x, double a0, double a1, double a2, double a3)
{ return a0 + (a1 + (a2 + a3*x)*x)*x; }
```

Esempio di cui sopra esteso:
{ FUNCTIONS/defaultvalues.cc }