Programmazione I - Laboratorio

Esercitazione 2 - Funzioni

Gianluca Mezzetti[1] Paolo Milazzo[2]

1. Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼mezzetti
mezzetti@di.unipi.it

2. Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼milazzo
milazzo@di.unipi.it

Corso di Laurea in Informatica
A.A. 2012/2013

Dichiarazione e definizione di funzioni

Le funzioni devono essere *dichiarate* e *definite*.

La dichiarazione viene fatta all'inizio ed ha questa forma:

```
tipo-restituito nome-di-funzione(parametri formali);
```

La definizione ha la forma

```
tipo-restituito nome-di-funzione(parametri formali){
    dichiarazioni
    istruzioni
}
```

provvedendo a far corrispondere i nomi della funzione e dei parametri formali. Esempio:

```
int stupida(int quanto, int perche);

int main(...){....}

int stupida(int quanto, int perche){
...
}
```

Le funzioni possono stare solamente top-level (non si possono definire funzioni dentro funzioni) !!

Valore restituito

Le funzioni possono restituire un valore attraverso il comando speciale **return** *espressione*.

```
int stupida(int quanto, int perche){
    int val;
    val = quanto / 2;
    return val;
}
```

Esiste un tipo speciale **void** per le funzioni che non restituiscono nulla.

```
void stupidissima(int quanto){
    ...
}
```

Chiamata

Una funzione viene chiamata usando il nome seguito dai parametri attuali tra parentesi, il numero ed il tipo dei parametri attuali deve corrispondere con quello dei parametri formali.

```
int x = 10;
stupida(x);
```

I parametri attuali vengono accoppiati con i parametri formali (il modo dipende dal meccanismo di *passaggio dei parametri*)
Quello che la funzione restituisce può essere ignorato o meno

```
int x = 10;
x = stupida(x);
```

Somma - somma.c

```c
/*
 Una funzione che somma due valori e restituisce il risultato
*/

#include<stdio.h>

int somma(int x, int y);

int main(void)
{
  int a,b;
  a = 2;
  b = 3;
  return somma(a,b);
}

int somma(int x, int y){
        return x + y;
}
```

Possiamo vedere il valore restituito dal main usando il comando *echo $?*

Passaggio di parametri - passaggiopervalore.c

Il passaggio di parametri in *C* avviene per *valore*.

```
/*
 Passaggio di parametri per valore
*/
#include<stdio.h>

void fun(int x);

int main(void)
{
        int x = 5;
        fun(x);
        printf("Dopo la chiamata x=%d \n",x);
        return 0;
}

void fun(int x){
        x=3;
}
```

Scoping - scoping.c

L'associazione tra i nomi e gli oggetti nell'ambiente (scoping) in C viene detta *lessicale* o *statica*. Essa dipende solo dalla struttura del programma piuttosto e non dal suo comportamento a tempo di esecuzione.

```
#include<stdio.h>

int fun(int x);

int x=3; /* scope globale */

int main(void)
{
        int y=8,x=5;
        printf("x dentro main %d \n",x);
        {
                int x=3,y=4;
                printf("x dentro un blocco %d \n",x);
                printf("y dentro un blocco %d \n",y);
        }
}
int fun(int x){
        /*int x=6;*/ /*NO: Ri-dichiarazione di variabile nel blocco
            */
        printf("x dentro fun %d \n",x);
        /*printf("y dentro fun %d \n",y);*/
  /*NO: Scoping lessicale, y non si vede*/
}
```

Ricorsione

Ogni funzione C può essere chiamata ricorsivamente
La definizione matematica della successione di Fibonacci è ricorsiva:

$$\begin{cases} fib_0 = 0 \\ fib_1 = 1 \\ fib_n = fib_{n-1} + fib_{n-2} & \text{if } n > 1 \end{cases}$$

Fibonacci ricorsivo - fibonacciricorsivo.c

```c
/*
 Fibonacci ricorsivo
*/

#include<stdio.h>

int fib(int n);

int main(void)
{
        printf("Fibonacci 0 : %d\n",fib(0));
        printf("Fibonacci 1 : %d\n",fib(1));
        printf("Fibonacci 2 : %d\n",fib(2));
        printf("Fibonacci 3 : %d\n",fib(3));
        printf("Fibonacci 20 : %d\n",fib(20));
        printf("Fibonacci 40 : %d\n",fib(40));
        printf("Fibonacci 42 : %d\n",fib(42));
        return 0;
}

int fib(int n){
        if(n==0)
                return 0;
        else if(n==1)
                return 1;
        else
                return fib(n-1) + fib(n-2);
}
```

Quante chiamate ? - fibonacciricorsivochiamate.c

```
#include<stdio.h>

int fib(int n);

/*->*/ int chiamate=0;

int main(void)
{
        printf("Fibonacci 0:%d,1:%d,2:%d\n",fib(0),fib(1),fib(2));
/*->*/  chiamate=0;
        printf("Fibonacci 20 : %d\n",fib(20));
/*->*/  printf("Effettuate %d chiamate\n",chiamate);chiamate=0;
        printf("Fibonacci 40 : %d\n",fib(40));
/*->*/  printf("Effettuate %d chiamate\n",chiamate);chiamate=0;
        printf("Fibonacci 42 : %d\n",fib(42));
/*->*/  printf("Effettuate %d chiamate\n",chiamate);
        return 0;
}

int fib(int n){
/*->*/  chiamate++;
        if(n==0)
                return 0;
        else if(n==1)
                return 1;
        else
                return fib(n-1) + fib(n-2);
}
```

Ricorsione vs Iterazione

- L'uso della ricorsione rende la stesura dei programmi più comprensibile e semplice ma è altrettanto semplice dare luogo a comportamenti non efficienti.
- Le chiamate a funzioni richiedono il mantenimento di una struttura dati aggiuntiva a runtime: i record di attivazione.
- Esiste un tipo di ricorsione che è equivalente all'iterazione perché non richiede di memorizzare i record di attivazione: *tail-recursion*.
- I compilatori in alcuni casi riconoscono la tail-recursion ed ottimizzano il codice trasformando la ricorsione in iterazione.

```
int tail(int x){
   ...
   return tail(...);
}
```

Fibonacci Ottimizzato e Tail Recursive

Notiamo che prendendo serie di coppie consecutive di valori di Fibonacci:

$$
\begin{aligned}
S_0 &= (fib_0, fib_1) = (0, 1) \\
S_1 &= (fib_1, fib_0 + fib_1) = (1, 1) \\
S_2 &= (fib_2, fib_1 + fib_2) = (1, 2) \\
S_3 &= (fib_3, fib_2 + fib_3) = (2, 3) \\
S_4 &= (fib_4, fib_3 + fib_4) = (3, 6) \\
&\vdots \\
S_i &= (fib_{i-1}, fib_{i-2} + fib_{i-1})
\end{aligned}
$$

possiamo dare una nuova definizione di Fibonacci (tenendo conto che S_{i+1} si può calcolare da S_i)

$$
\begin{cases}
fib_{0,S_i} = first(S_i) \\
fib_{n,S_i} = fib_{n-1,S_{i+1}}
\end{cases}
$$

tale che $fib_n = fib_{n,S_0}$

Fibonacci Ottimizzato e Tail Recursive - fibonaccitailrecursive.c

```c
/*
 Fibonacci tail-recursive
*/

#include<stdio.h>

int fib(int n);
int fibt(int n, int first, int second);

int main(void)
{
        printf("Fibonacci 42 : %d\n",fib(42));
        return 0;
}

int fib(int n){return fibt(n,0,1);}
int fibt(int n, int first, int second){
        if(n==0)
                return first;
        else
                return fibt(n-1,second,first+second);
}
```

Quante chiamate ? - fibonaccitailrecursivechiamate.c

```
#include<stdio.h>

int fib(int n);
int fibt(int n, int first, int second);

int chiamate=0;

int main(void)
{
        printf("Fibonacci 42 : %d\n",fib(42));
        printf("Effettuate %d chiamate\n",chiamate);
        return 0;
}

int fib(int n){return fibt(n,0,1);}
int fibt(int n, int first, int second){
        chiamate++;
        if(n==0)
                return first;
        else
                return fibt(n-1,second,first+second);
}
```

Radice quadrata

Realizzare una funzione che calcoli la radice quadrata di un numero

- La radice quadrata di un numero x è compresa tra 0 e x
- Se $(x/2)^2 > x$ allora la radice è tra 0 e $x/2$, altrimenti è tra $x/2$ e x
- Si può proseguire prendendo il punto centrale tra $x/2$ e x o 0 ed $x/2$

Radice quadrata - radice.c I

```
/*
 Una funzione che calcola la radice quadrata
*/
#include<math.h> /*for fabs() function*/
#include<stdio.h>

#define PRECISIONE 0.0001

float radice(float x);
float radicerc(float x,float low,float hi);

int main(void)
{
        printf("La radice di 5 e' %f",radice(2));
        return 0;
}

float radice(float x){
        if(x<=0)
                return -1;
        else
                return radicerc(x,0,x);
}
float radicerc(float x,float low,float hi){
        float mid;
        mid = (hi+low)/2.0;
        /*printf("mid:%f \n",mid); */
```

Radice quadrata - radice.c II

```
        if(fabs(mid*mid - x) < PRECISIONE)
                return mid;
        else if (mid*mid - x > 0)
                return radicerc(x,low,mid);
        else
                return radicerc(x,mid,hi);
}
```

Esercizio: massimo divisore comune

Calcoliamo il massimo divisore comune (G.C.D.) tra due interi a e b positivi (> 0).

- L'algoritmo di Euclide si basa sull'osservazione che il G.C.D. tra due numeri divide anche la loro differenza ($a = xm, b = xn$ allora $a - b = x(m - n)$).
- Quindi è possibile calcolare il G.C.D. tra due numeri calcolando quello tra uno di loro e la differenza tra i due.

Esercizio: massimo divisore comune

Calcoliamo il massimo divisore comune (G.C.D.) tra due interi a e b positivi (> 0).

- L'algoritmo di Euclide si basa sull'osservazione che il G.C.D. tra due numeri divide anche la loro differenza ($a = xm, b = xn$ allora $a - b = x(m - n)$).
- Quindi è possibile calcolare il G.C.D. tra due numeri calcolando quello tra uno di loro e la differenza tra i due.

Suggerimento:

$$
\begin{aligned}
gcd(a, a) &= a \\
gcd(a, b) &= gcd(a, b - a) \quad \text{if } a < b \\
gcd(a, b) &= gcd(a - b, b) \quad \text{if } a > b
\end{aligned}
$$

Esercizio: massimo divisore comune - gcd.c

```
/*
 Massimo divisore comune
*/
#include<stdio.h>

int gcd(int a, int b);

int main(void)
{
        printf("GCD tra 5 e 10 e' %d \n",gcd(5,10));
        printf("GCD tra 30 e 35 e' %d \n",gcd(30,35));
        printf("GCD tra 13 e 17 e' %d \n",gcd(13,17));
        printf("GCD tra 20384572 e 356373848 e' %d \n",gcd
            (20384572,356373848));
        return 0;
}

int gcd(int a, int b){
        if(a==b)
                return a;
        else if(a < b)
                return gcd(a,b-a);
        else
                return gcd(a-b,b);
}
```

Esercizio: Funzione di Ackermann

Definire la seguente funzione

$$A(z,x,y) = \begin{cases} y & \text{se } z = 0, x = 0 \\ A(0, x-1, y) + 1 & \text{se } z = 0, x \geq 1 \\ 0 & \text{se } z = 1, x = 0 \\ 1 & \text{se } z \geq 2, x = 0 \\ A(z-1, A(z, x-1, y), y) & \text{se } z \geq 1, x \geq 1 \end{cases}$$

Il programma inoltre deve calcolare il numero di chiamate alla funzione A e stamparlo.

Esercizio: Funzione di Ackermann - ackermann.c I

```
#include<stdio.h>

int A(int z, int x, int y);

int chiamate=0;

int main(void)
{
        printf("A(0,0,0) = %d\n",A(0,0,0));
        printf("Effettuate %d chiamate\n",chiamate);chiamate=0;

        printf("A(0,3,4) = %d\n",A(0,3,4)); /*A(0,x,y)=y+x*/
        printf("Effettuate %d chiamate\n",chiamate);chiamate=0;

        printf("A(1,3,4) = %d\n",A(1,3,4)); /*A(1,x,y)=yx*/
        printf("Effettuate %d chiamate\n",chiamate);chiamate=0;

        printf("A(2,3,4) = %d\n",A(2,3,4)); /*A(2,x,y)=y^x*/
        printf("Effettuate %d chiamate\n",chiamate);chiamate=0;

        printf("A(3,3,4) = %d\n",A(3,3,4)); /*4^(4^4)*/
        printf("Effettuate %d chiamate\n",chiamate);chiamate=0;

        /*
        f(z,u,v) definisce una operazione binaria tra u e v al loro
             variare
        f(z+1,x,y) applica f(z,u,v) ad y x-1 volte.
```

Esercizio: Funzione di Ackermann - ackermann.c II

```
        cioe' f(z,f(z+1,x-1,y),y)
        */
        return 0;
}

int A(int z, int x, int y){
        chiamate++;
        if (z==0 && x==0) return y;
        else if (z==0 && x>=0) return A(0,x-1,y)+1;
        else if (z==1 && x==0) return 0;
        else if (z>=2 && x==0) return 1;
        else return A(z-1,A(z,x-1,y),y);
}
```

Esercizio: Probabilità di avere una buona giornata (meglio al PC)

In Russia i biglietti degli autobus sono numerati con numeri a 6 cifre. Si dice che un russo possa augurarsi una buona giornata se quando sale sul bus la mattina possiede un biglietto in cui somma delle prima 3 cifre è la stessa delle ultime 3.
Sviluppare un programma che calcoli la probabilità di avere una buona giornata

- Definire una funzione *fortunato* che verifica se un biglietto a 6 cifre è fortunato o meno.
- Contare il numero di biglietti fortunati totali e calcolare la probabilità (biglietti fortunati / biglietti totali)

Calcolare la probabilità per biglietti da $2, 4, 6, 8, 10$ cifre e concludere quindi quale sia il numero migliore di cifre perché i russi siano si sentano più fortunati.

```c
/*
  Probabilita' di avare una buona giornata
*/
#include <stdio.h>

int fortunato(int biglietto,int cifre);

int potenza(int base, int exp);

int main()
{
  int i;
  int casi_possibili;
  int casi_favorevoli;
  float p;
  int cifre;

  /* printf("fortunato 123321 = %d\n" ,fortunato(123321)); */

  for (cifre=2; cifre<=10; cifre=cifre+2)
  {
    casi_possibili = potenza(10,cifre);

    casi_favorevoli=0;
    for (i=0; i<casi_possibili; i++)
    {
      if (fortunato(i,cifre)) casi_favorevoli++;
      /* ALTERNATIVA: casi_favorevoli = casi_favorevoli + fortunato
          (i); */
```

```
    }

    p = ((float) casi_favorevoli/(float) casi_possibili);
    printf("Con %d cifre probabilita' %6.3f (%d casi su %d)\n",
        cifre,p,casi_favorevoli,casi_possibili);
  }

}

int fortunato(int biglietto,int cifre)
{
  int i;
  int somma_prime, somma_ultime;

  somma_ultime = 0;
  for (i=0; i<cifre/2; i++)
  {
    somma_ultime = somma_ultime + biglietto%10;
    biglietto = biglietto/10;
  }

  somma_prime = 0;
  for (i=0; i<cifre/2; i++)
  {
    somma_prime = somma_prime + biglietto%10;
    biglietto = biglietto/10;
  }

  return (somma_ultime==somma_prime);
}
```

```
int potenza (int base, int exp)
{
  int i;
  int risultato=1;

  for (i=0; i<exp; i++)
  {
    risultato = risultato*base;
  }

  return risultato;
}
```