Funzioni

Libro cap. 5

Moreno Marzolla
Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
http://www.moreno.marzolla.name/

Ringraziamenti

- prof. Stefano Mizzaro, Università di Udine
  - http://users.dimi.uniud.it/~stefano.mizzaro/

Funzione

- Intuitivamente definita come "*un blocco di istruzioni con un nome*"
- Può accettare un certo numero di parametri (anche nessuno)
- Può restituire un risultato, oppure nessun risultato

```
/* demo-funzioni.c */
#include <stdio.h>

double somma(double a, double b)
{
    double s = a + b;
    return s;
}

void stampa(double v)
{
    printf("v=%f\n", v);
    return; /* opzionale */
}

void saluta( void )
{
    printf("Ciao\n");
}

int main( void )
{
    saluta();
    stampa(somma(3.0, 4.0/2.0));
    return 0;
}
```

Funzione

- Se la funzione ritorna "void" (cioè nessun risultato)
  - il **`return`** è facoltativo
  - se manca, la funzione termina quando l'esecuzione arriva in fondo al corpo
- Se la funzione ritorna un valore
  - è obbligatorio **`return`** ***`expr`***
  - *expr* è una espressione il cui valore è quello che verrà restituito dalla funzione

```c
/* demo-funzioni.c */
#include <stdio.h>

double somma(double a, double b)
{
    double s = a + b;
    return s;
}

void stampa(double v)
{
    printf("v=%f\n", v);
    return; /* opzionale */
}

void saluta( void )
{
    printf("Ciao\n");
}

int main( void )
{
    saluta();
    stampa(somma(3.0, 4.0/2.0));
    return 0;
}
```

Dichiarazione vs Definizione

- La dichiarazione di una funzione indica la sua *segnatura*
  - Nome della funzione; tipo del risultato; tipo dei parametri
  - Il corpo della funzione può essere omesso
  - Una funzione può essere usata solo dopo che è stata dichiarata
- La definizione di una funzione indica *cosa fa*
  - Il corpo della funzione deve essere specificato
  - La definizione funge anche da dichiarazione

Dichiarazione vs Definizione

```c
/* dich.c */
#include <stdio.h>

int fattoriale(int);

int fat5( void ) { return fattoriale(5); }

int fattoriale(int n)
{
    int f = 1;
    while (n > 1) {
        f = f*n;
        n--;
    }
    return f;
}

int main( void )
{
    printf("%d\n", fat5() );
    return 0;
}
```

***Dichiarazione** della funzione "fattoriale" che accetta un parametro intero e restituisce un intero.*

La funzione fattoriale() può essere usata dopo che è stata dichiarata

***Definizione** della funzione fattoriale*

Definizione delle funzioni

- Al di fuori del **`main`** e di altre funzioni
- Intestazione (prima riga)
  - Tipo del risultato restituito (**`void`** se non restituisce un risultato)
  - Nome della funzione
  - Parametri e loro tipi
    - separati da ","
    - se omessi si assume **`void`**
- Corpo (fra graffe { } obbligatorie)
  - Istruzioni
  - **`return`** termina esecuzione della funzione
    - **`return`** ***`expr`*** è obbligatorio per funzioni che restituiscono un valore
    - Per quelle che *non* restituiscono un valore si può usare "return" senza argomento, oppure ometterlo per terminare la funzione alla fine della funzione

Esecuzione di programmi con funzioni

- L'esecuzione comincia da main
  - Anche il **`main`** è una funzione
- Invocazione → l'esecuzione passa alla funzione
  - Associazione parametri
  - Esecuzione istruzioni della funzione
  - … fino al **`return`** o alla fine della funzione
- Poi l'esecuzione ritorna al chiamante, all'istruzione successiva

Flusso di esecuzione

```c
/* dich.c */
#include <stdio.h>

int fattoriale(int);

int fat5( void )
{ return fattoriale(5); }

int fattoriale(int n)
{
  int f = 1;
  while (n > 1) {
    f = f*n;
    n--;
  }
  return f;
}

int main( void )
{
  printf("%d\n", fat5() );
  return 0;
}
```

```c
int fat5( void )
{
  return fattoriale(5);
}
```

```c
int fattoriale(int n)
{
  int f = 1;
  while (n > 1) {
    f = f*n;
    n--;
  }
  return f;
}
```

Parametri formali/attuali

```
/* cotangente.c */
#include <stdio.h>
#include <math.h>

double tangente(double x) {
    return (sin(x) / cos(x));
}

double cotangente (double x) {
    return (1.0 / tangente(x));
}

int main( void )
{
    double angolo;
    printf("Inserisci angolo ");
    scanf("%lf", &angolo);
    printf("La cotangente di %f e' %f\n", angolo,
            cotangente(angolo));
    return 0;
}
```

*Parametro **formale**: usato nella definizione della funzione*

*Parametro **attuale**: specifica su quali parametri la funzione deve operare*

Cosa succede?

```c
/* parametri.c */
#include <stdio.h>

void inc(int x)
{
    x = x + 1;
}

int main( void )
{
    int y = 0;
    inc(y);
    printf("%d\n", y);
    return 0;
}
```

Output

```
0
```

Perché

- Il passaggio di parametri avviene per valore
- Il valore del parametro attuale viene copiato nel parametro formale
  - Ma sono due variabili diverse
  - Quindi ogni modifica al parametro formale **non** si riflette sul parametro attuale
- Vedremo più avanti il meccanismo da usare perché una funzione possa “modificare” i parametri attuali

Esempio

- Scrivere una funzione che determina se un numero **`n`** positivo è un numero primo oppure no
- La funzione riceve come parametro un intero **`n`** e restituisce **`1`** se **`n`** è primo, **`0`** altrimenti
  - 1 non è un numero primo
  - 2 è un numero primo
- Scrivere anche un **`main()`** che invoca la funzione in modo appropriato

Idea

- Guardo tutti i numeri **`i`** minori di **`n`**
  - Escluso 1
  - Basta arrivare a $\sqrt{\texttt{n}}$
- Controllo per ogni **`i`** se è un divisore di **`n`**
- Se trovo un divisore di **`n`**, allora **`n`** non è primo
  - Ne basta uno; mi fermo subito
- Se non ne trovo, **`n`** è primo
  - Devo guardarli tutti; esco solo alla fine

SBAGLIATO

```c
#include <stdio.h>

/* Dato un intero n > 0, restituisce 1 se n e' primo, 0 altrimenti */
int primo(int n)
{
    int i;
    for (i = 2; i*i <= n; i++) {
      if (n % i == 0) {
        return 0;
      } else {
        return 1;
      }
    }
}

int main( void )
{
   int v;
   printf("Inserisci un intero "); scanf("%d", &v);
   if ( primo(v) ) {
      printf("%d e' un numero primo\n", v);
   } else {
      printf("%d NON e' un numero primo\n", v);
   }
   return 0;
}
```

SBAGLIATO

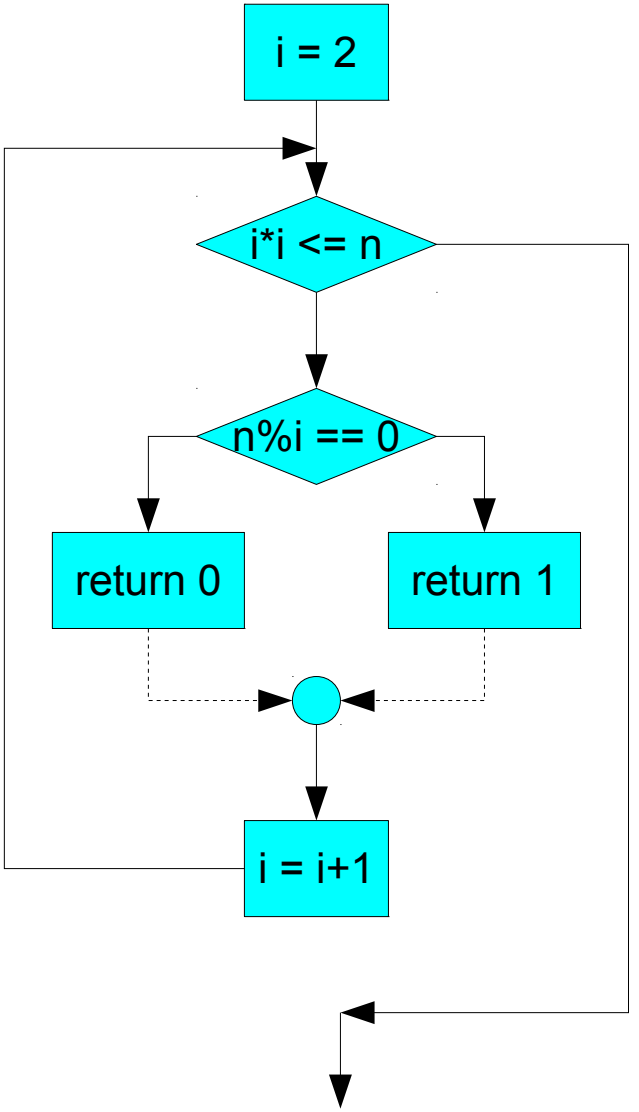
```c
#include <stdio.h>

/* Dato un intero n > 0, restituisce 1 se n e' primo, 0 altrimenti */
int primo(int n)
{
     int i;
     for (i = 2; i*i <= n; i++) {
       if (n % i == 0) {
         return 0;
       } else {
         return 1;
       }
     }
}

int main( void )
{
    int v;
    printf("Inserisci un intero "); scanf("%d", &v);
    if ( primo(v) ) {
        printf("%d e' un numero primo\n", v);
    } else {
        printf("%d NON e' un numero primo\n", v);
    }
    return 0;
}
```

Il compilatore potrebbe aiutarci

- ...purché abilitiamo tutti i messaggi di avvertimento (warning) come mostrato in precedenza

```
primo.c: In function ‘primo’:
primo.c:14:1: warning: control reaches end of non-void function [-
   Wreturn-type]
 }
 ^
```

```
int primo(int n)
{
    int i;
    for (i = 2; i*i <= n; i++) {
      if (n % i == 0) {
        return 0;
      } else {
        return 1;
      }
    }
}
```



Meglio, ma sempre sbagliato

```
#include <stdio.h>

/* Dato un intero n maggiore di zero, restituisce 1 se n e' primo, 0 altrimenti */
int primo(int n)
{
    int i;
    for (i = 2; i*i <= n; i++) {
      if (n % i == 0) {
        return 0; /* trovato un divisore; n non primo */
      }
    }
    return 1; /* nessun divisore; n e' primo */
}

int main( void )
{
   int v;
   printf("Inserisci un intero "); scanf("%d", &v);
   if ( primo(v) ) {
      printf("%d e' un numero primo\n", v);
   } else {
      printf("%d NON e' un numero primo\n", v);
   }
   return 0;
}
```

Test

- Verifichiamo empiricamente la correttezza della funzione **`primo(n)`**, testando i “casi limite”
  - n = 1, n = 2

SBAGLIATO

```
int primo(int n)
{
    int i;
    for (i = 2; i*i <= n; i++) {
      if (n % i == 0) {
        return 0;    /* n non e' primo */
      }
    }
    return 1;        /* n e' primo */
}
```

Versione corretta

- Mettiamoci “una pezza”

OK

```
/* primo.c */
int primo(int n)
{
  int i;
  if ( n == 1 ) {
    return 0;
  } else {
    for (i = 2; i*i <= n; i++) {
      if (n % i == 0) {
        return 0;
      }
    }
    return 1;
  }
}
```

Il meccanismo di invocazione delle funzioni

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

x = 3, y = 2, sqx = ?, sqy = ?

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

Esempio

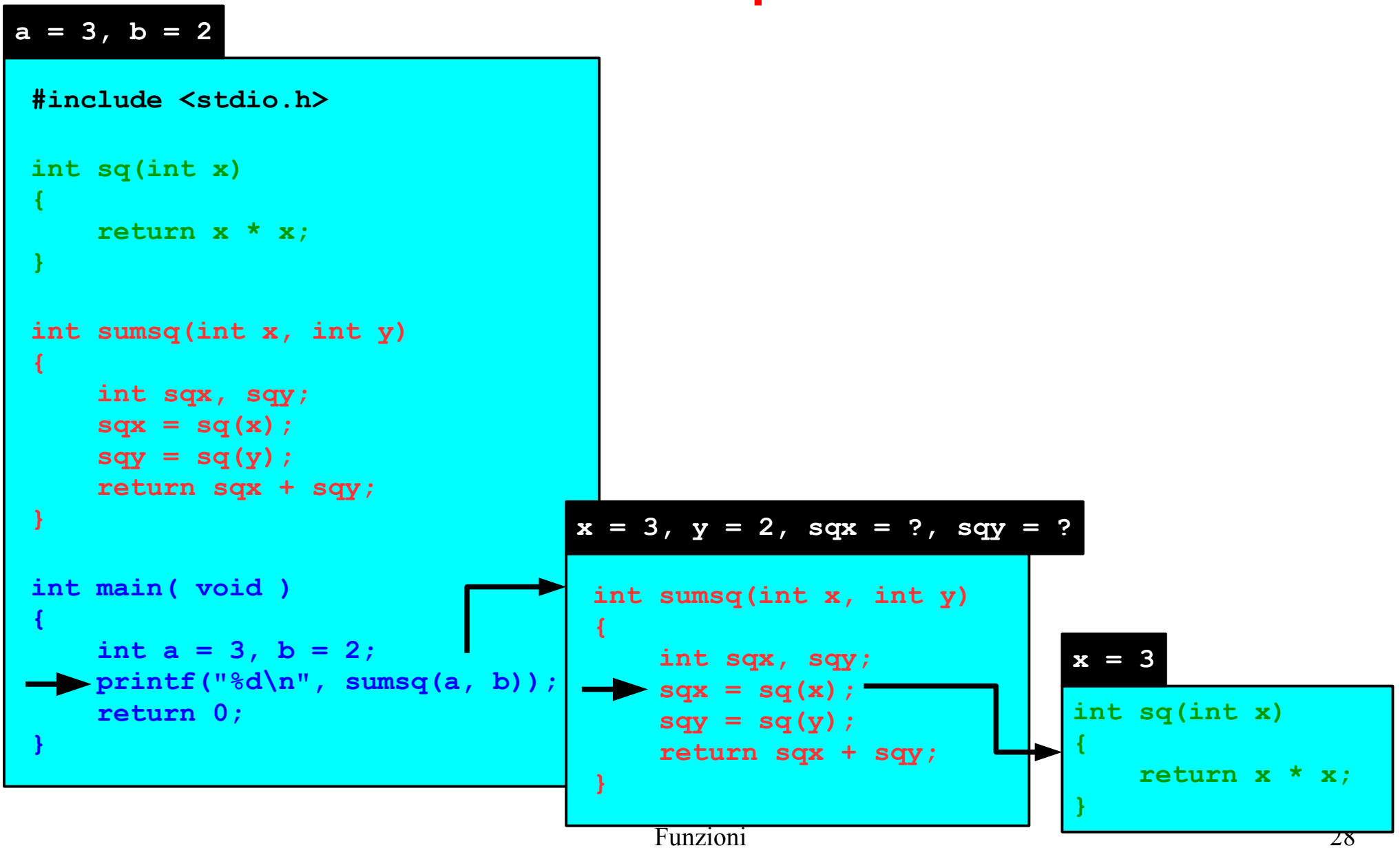
a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

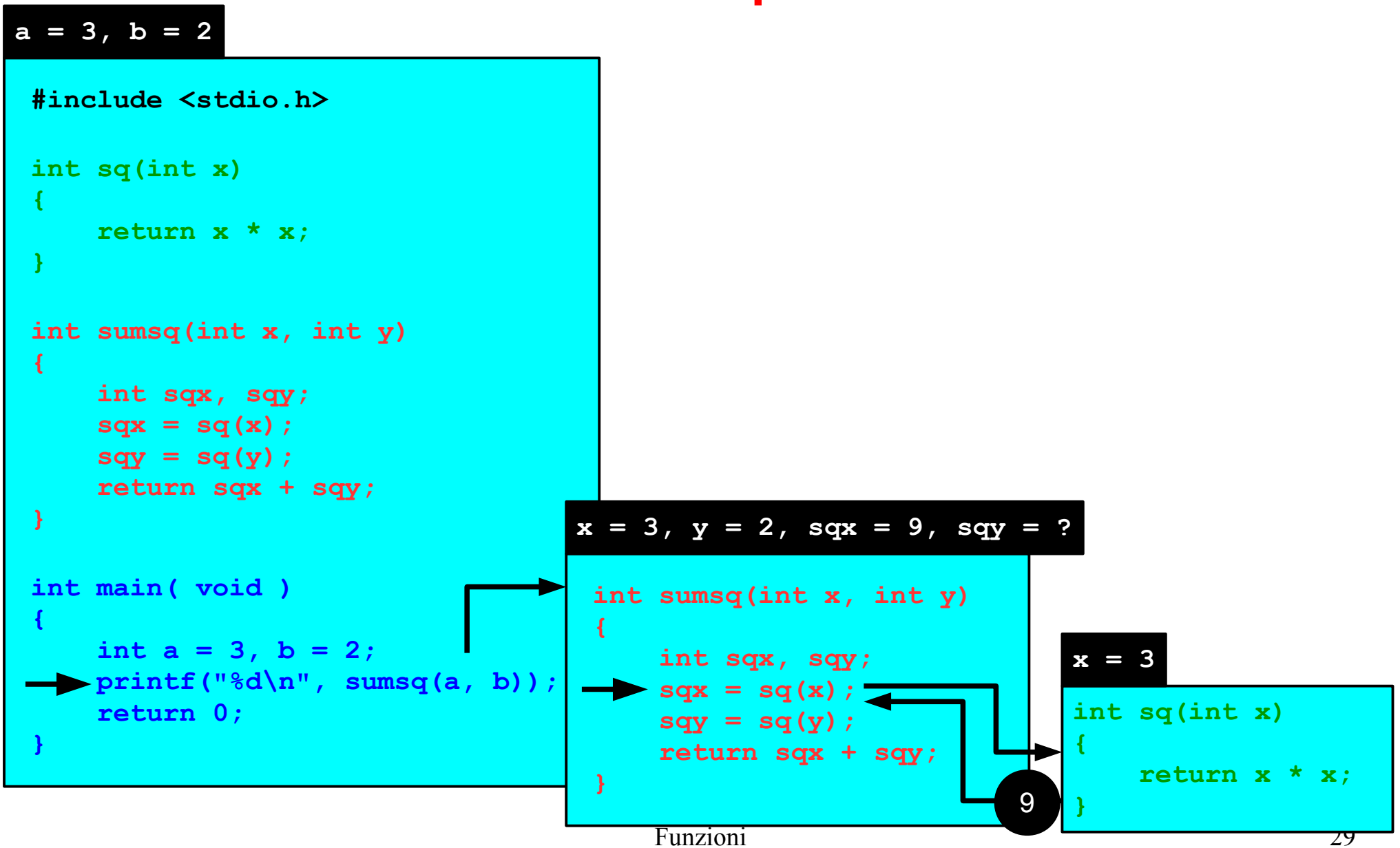
x = 3, y = 2, sqx = ?, sqy = ?

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

Esempio


a = 3, b = 2
#include <stdio.h>
int sq(int x)
{
return x * x;
}
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
int main(void)
{
int a = 3, b = 2;
printf("%d\n", sumsq(a, b));
return 0;
}
x = 3, y = 2, sqx = ?, sqy = ?
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
x = 3
int sq(int x)
{
return x * x;
}

Esempio


a = 3, b = 2
#include <stdio.h>
int sq(int x)
{
return x * x;
}
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
int main(void)
{
int a = 3, b = 2;
printf("%d\n", sumsq(a, b));
return 0;
}
x = 3, y = 2, sqx = 9, sqy = ?
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
x = 3
int sq(int x)
{
return x * x;
}
9

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

x = 3, y = 2, sqx = 9, sqy = ?

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

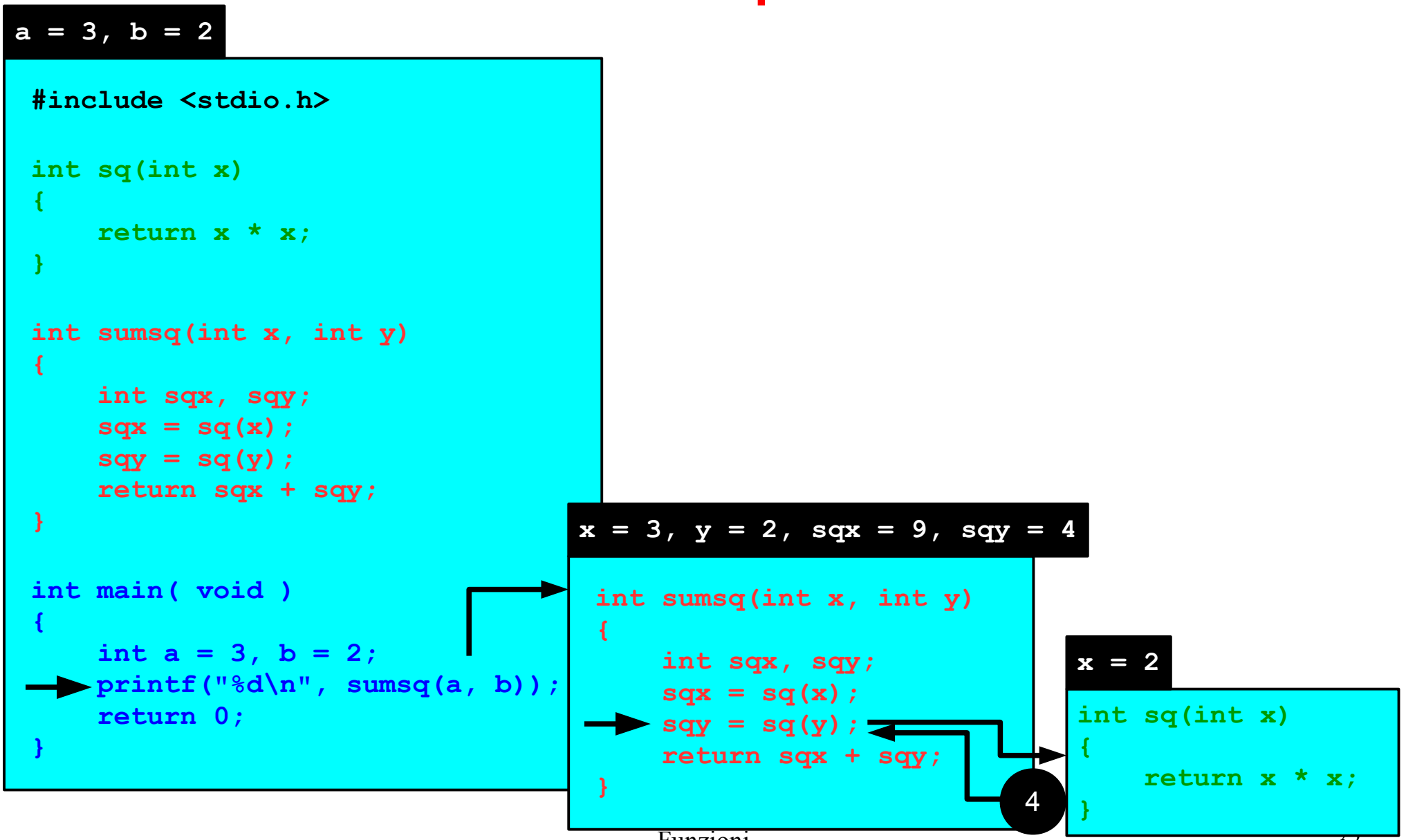
x = 3, y = 2, sqx = 9, sqy = ?

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

x = 2

```
int sq(int x)
{
    return x * x;
}
```

Esempio


a = 3, b = 2
#include <stdio.h>
int sq(int x)
{
return x * x;
}
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
int main(void)
{
int a = 3, b = 2;
printf("%d\n", sumsq(a, b));
return 0;
}
x = 3, y = 2, sqx = 9, sqy = 4
int sumsq(int x, int y)
{
int sqx, sqy;
sqx = sq(x);
sqy = sq(y);
return sqx + sqy;
}
4
x = 2
int sq(int x)
{
return x * x;
}

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

x = 3, y = 2, sqx = 9, sqy = 4

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

13

x = 3, y = 2, sqx = 9, sqy = 4

```
int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}
```

Esempio

a = 3, b = 2

```
#include <stdio.h>

int sq(int x)
{
    return x * x;
}

int sumsq(int x, int y)
{
    int sqx, sqy;
    sqx = sq(x);
    sqy = sq(y);
    return sqx + sqy;
}

int main( void )
{
    int a = 3, b = 2;
    printf("%d\n", sumsq(a, b));
    return 0;
}
```

Visibilità delle variabili

Regola di visibilità delle variabili

- Una variabile è visibile unicamente all'interno del blocco in cui è dichiarata
  - I parametri formali funzionano come le variabili (sono visibili solo nel corpo della funzione cui si riferiscono)
- Blocco: da { a }
- Le variabili globali sono dichiarate al di fuori da ogni blocco e sono accessibili ovunque
  - In qualunque punto del main() e di qualunque altra funzione

Modello a contorni

- Utile per capire dove una variabile è visibile
- Un “contorno” (rettangolo) intorno a ogni blocco
- All'interno di un blocco sono visibili
  - Tutte le variabili definite all'inizio di quel blocco
  - Tutte le variabili definite nei blocchi che contengono quel blocco

Modello a contorni: esempio

```
int va;                                                    va

void f(int vb) {                                           vb
    ...
    for (...) {
        int vc;                                            vc
        if (...) {
            int vd;                                        vd
            ...
        }
    }
}

int main( void )
{
    int ve;                                                ve
    ...
}
```


.....

Blocchi annidati

- È possibile (ma sconsigliato) definire variabili che hanno lo stesso nome di variabili già definite in un blocco più esterno
- Le variabili definite nel blocco interno “nascondono” quelle con lo stesso nome definite esternamente
  - Vedi esempio che segue

Esempio

```
/* nascoste.c */
#include <stdio.h>

int x, y;     /* variabili globali */

void f( void )
{
    int x;
    x = 1;     /* locale */
    y = 1;     /* globale */
    printf("f(): x=%d\n", x);
    printf("f(): y=%d\n", y);
}

int main( void )
{
    x = 0;     /* globale */
    y = 0;     /* globale */
    f();
    printf("main(): x=%d\n", x);
    printf("main(): y=%d\n", y);
    return 0;
}
```

x, y

x

```
f(): x=1
f(): y=1
main(): x=0
main(): y=1
```

Variabili globali vs. passaggio parametri

- Quando possibile, usare parametri e non variabili globali
- È più semplice capire cosa fa una funzione
  - Basta leggere la definizione della funzione; non serve far riferimento a tutto il resto del programma

Alcune funzioni matematiche dichiarate in `<math.h>`

- **`double sqrt(double)`**
  - Radice quadrata
- **`double exp(double x)`**, **`double exp2(double x)`**, **`double exp10(double x)`**
  - *e* (base dei logaritmi naturali) elevato alla *x*, 2 elevato alla *x*, 10 elevato alla *x*
- **`double log(double x)`**, **`double log2(double x)`**, **`double log10(double x)`**
  - logaritmo in base *e*, in base 2, in base 10 di *x*
- **`double sin(double x)`**, **`double cos(double x)`**, **`double tan(double x)`**
  - seno, coseno, tangente di *x*
- **`double pow(double x, double y)`**
  - *x* elevato alla *y*
- **`double fabs(double x)`**
  - valore assoluto di *x*
- **`double floor(double x)`**, **`double ceil(double x)`**
  - arrotonda *x* all'intero inferiore o superiore

Uso di <math.h>

- Le funzioni di cui al lucido precedente (e molte altre) sono dichiarate in **math.h**

```
/* esempio-math.c : esempio di uso di alcune funzioni matematiche */
#include <stdio.h>
#include <math.h>

int main( void )
{
    double v;
    for (v=0.0; v < 1.0; v += 0.1) {
        printf("%f %f %f %f\n", v, sin(v), cos(v), tan(v));
    }
    return 0;
}
```

Esercizio

- Scrivere una funzione **`int primofattore(int n)`** che, dato in input un intero $n \geq 1$, restituisce il più piccolo fattore primo di i; se n è un numero primo, la funzione restituisce n
  - Es.: primofattore(1) restituisce 1; primofattore(2) restituisce 2; primofattore(4) restituisce 2; primofattore(9) restituisce 3; primofattore(21) restituisce 3
  - Suggerimento: bastano alcune modifiche alla funzione primo()
- Scrivere una funzione **`void fattorizzazione(int n)`** che stampa la fattorizzazione di un intero $n \geq 1$
  - Es.: fattorizzazione(1) stampa 1; fattorizzazione(3) stampa 3; fattorizzazione(21) stampa 3 7; fattorizzazione(16) stampa 2 2 2 2; fattorizzazione(18) stampa 2 3 3
  - Suggerimento: si può sfruttare la funzione primofattore(), oppure sviluppare una nuova soluzione ad hoc