Capitolo 5 - Funzioni

Outline

Introduzione
Moduli in C
Math Library Functions
Funzioni
Definizione di Funzioni
Prototipi di Funzione
Header Files
Chiamata di funzioni per valore e referenza
Random Number Generation
Storage Classes
Scope Rules
Ricorsione
La serie di Fibonacci
Ricorsione vs. Iterazione

Introduzione

- Divide and conquer
  - Costruire un programma da pezzi più piccoli o da singole componenti
    - Questi pezzi più piccoli sono chiamati moduli
  - Ogni singolo pezzo è più facilmente gestibile rispetto al programma principale

Moduli in C

- Functions
  - Moduli in C
  - Il programma combina funzioni user-defined con funzioni di libreria
    - La libreria standard del C ha una grande varietà di funzioni
- Chiamata di funzioni
  - Invocazione
    - Si fornisce il nome della funzione e gli argomenti (dati)
    - La funzione esegue le operazioni opportune
    - La funzione restituisce un risultato

Math Library Functions

- Math library functions
  - Esegue comuni calcolo matematici
  - `#include <math.h>`
- Formato per la chiamata di funzioni
  - `FunctionName( argument );`
    - Se ci sono più argomenti, si usa una lista separata da virgole
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Chiama la funzione `sqrt`, che restituisce la radice quadrata dell'argomento
    - Tutte le funzioni matematiche restituiscono dati di tipo `double`
  - Gli argomenti possono essere costanti, variabili o espressioni

Math Library Functions

Function	Description	Example
`sqrt( x )`	square root of *x*	`sqrt( 900.0 ) is 30.0` `sqrt( 9.0 ) is 3.0`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282` `exp( 2.0 ) is 7.389056`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 ) is 1.0` `log( 7.389056 ) is 2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 1.0 ) is 0.0`
		`log10( 10.0 ) is 1.0`
		`log10( 100.0 ) is 2.0`

`fabs( x )`	absolute value of *x*	`fabs( 5.0 ) is 5.0`
		`fabs( 0.0 ) is 0.0`
		`fabs( -5.0 ) is 5.0`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 ) is 128.0` `pow( 9, .5 ) is 3.0`
`fmod( x, y )`	remainder of *x*/*y* as a floating point number	`fmod( 13.657, 2.333 ) is 1.992`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 ) is 0.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 ) is 1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 ) is 0.0`

Fig. 5.2 Commonly used math library functions.

Funzioni

- Funzioni
  - Modularizzano un programma
  - Tutte le variabili definite all'interno delle funzioni sono locali
    - Conosciute solo nella funzione in cui sono state definite
  - Parametri
    - Permettono la comunicazione tra funzioni
    - Sono anch'essi variabili locali
- Benefici delle funzioni
  - Divide and conquer
  - Software reusability
    - Uso di funzioni esistenti come building blocks per nuovi programmi
    - Astrazione – nascondono i dettagli interni
  - Evitano la ripetizione del codice

Definizione di funzioni

- Formato per la definizione di funzioni

```
return-value-type  function-name( parameter-list )
{
   declarations and statements
}
```

  - Function-name: ogni identificatore valido
  - Return-value-type: tipo del risultato (default `int`)
    - `void` – indica che la funzione non restituisce nulla
  - Parameter-list: lista separata da virgole che permette di dichiarare i parametri
    - Specificare il tipo per ogni parametro

Definizione di funzioni

- Formato per la definizione di funzioni (continua)

```
return-value-type  function-name( parameter-list )
{
   declarations and statements
}
```

  - Declarations and statements: corpo della funzione (block)
    - Le variabili possono essere definite all'interno del blocco (possono essere innestati)
    - Le funzioni non possono essere definite all'interno di altre funzioni (non è permesso l'annidamento)
  - Ritorno del controllo
    - Se non deve essere restituito nulla all'esterno
      - `return;`
      - oppure si raggiunge la parentesi di chiusura
    - Se qualcosa deve essere restituito
      - `return` *expression* `;`

```c
/* Fig. 5.3: fig05_03.c
   Creating and using a programmer-defined function */
#include <stdio.h>

int square( int y );   /* function prototype */

/* function main begins program execution */
int main()
{
   int x; /* counter */

   /* loop 10 times and calculate and output square of x each time */
   for ( x = 1; x <= 10; x++ ) {
      printf( "%d ", square( x ) ); /* function call */
   } /* end for */

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

Outline

fig05_03.c (Part 1 of 2)


```c
/* square function definition returns square of an integer */
int square( int y ) /* y is a copy of argument to function */
{
   return y * y; /* returns square of y as an int */

} /* end function square */
```

```
1  4  9  16  25  36  49  64  81  100
```

Outline

fig05_03.c (Part 2 of 2)

Program Output


```c
/* Fig. 5.4: fig05_04.c
   Finding the maximum of three integers */
#include <stdio.h>

int maximum( int x, int y, int z ); /* function prototype */

/* function main begins program execution */
int main()
{
   int number1; /* first integer */
   int number2; /* second integer */
   int number3; /* third integer */

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &number1, &number2, &number3 );

   /* number1, number2 and number3 are arguments
      to the maximum function call */
   printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );

   return 0; /* indicates successful termination */

} /* end main */
```

Outline

fig05_04.c (Part 1 of 2)


```c
/* Function maximum definition */
/* x, y and z are parameters */
int maximum( int x, int y, int z )
{
   int max = x;      /* assume x is largest */

   if ( y > max ) { /* if y is larger than max, assign y to max */
      max = y;
   } /* end if */

   if ( z > max ) { /* if z is larger than max, assign z to max */
      max = z;
   } /* end if */

   return max;       /* max is largest value */

} /* end function maximum */
```

```
Enter three integers: 22 85 17
Maximum is: 85
Enter three integers: 85 22 17
Maximum is: 85
Enter three integers: 22 17 85
Maximum is: 85
```

Outline

fig05_04.c (Part 2 of 2)

Program Output

Prototipi di funzione

- Prototipi di funzione
  - Tipo di ritorno (default `int`)
  - Nome della funzione
  - Parametri di input
  - Usati per validare le funzioni
  - I prototipi sono necessari solo quando la definizione delle funzioni avviene dopo l'uso nel programma
  - La funzione con il seguente prototipo

```
int maximum( int x, int y, int z );
```

    - Ha tre parametri di input interi
    - Restituisce un `int`

Header Files

- Header files
  - Contengono i prototipi delle funzioni per le funzioni di libreria
  - `<stdlib.h>`, `<math.h>`, etc…
  - Caricati con `#include <filename>`

```
#include <math.h>
```

- Custom header files
  - Creare file con delle funzioni
  - Salvare il file come `filename.h`
  - Caricarlo in altri file con `#include "filename.h"`
  - Riuso di funzioni

Header Files

Standard library header	Explanation
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common definitions of types used by C for performing certain calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Fig. 5.6 Some of the standard library header.

Chiamata di funzioni per valore e riferimento

- Chiamata per valore
  - Copia un argomento passato ad una funzione
  - I cambiamenti nella funzione non hanno effetto sui parametri originali
  - Usare quando la funzione non ha bisogno di modificare degli argomenti
    - Evita modifiche accidentali
- Chiamata per riferimento (o indirizzo)
  - Passa l'indirizzo di memoria in cui si trovano gli argomenti originali
  - I cambiamenti nella funzione hanno effetto sui parametri originali
- Per ora focalizziamoci solo sulle chiamate per valore

Random Number Generation

- `rand` function
  - `<stdlib.h>`
  - Restituisce un numero "random" tra 0 and `RAND_MAX` (al massimo `32767`)

    ```
    i = rand();
    ```
  - Pseudorandom
    - Sequence predefinita di numeri "random"
    - Stessa sequenza per ogni chiamata della funzione
- Scaling
  - Per avere un numero random tra 1 and `n`

    ```
    1 + ( rand() % n )
    ```
    - `rand() % n` restituisce un numero tra 0 e `n - 1`
    - Aggiungere 1 per rendere il numero tra 1 ed `n`

      ```
      1 + ( rand() % 6)
      ```
      - numeri tra 1 e 6

Random Number Generation

- `srand` function
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**
  - `srand( time( NULL ) );/*load <time.h> */`
    - `time( NULL )`
      - Returns the time at which the program was compiled in seconds
      - "Randomizes" the seed

```
1  /* Fig. 5.7: fig05_07.c
2     Shifted, scaled integers produced by 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main()
8  {
9     int i; /* counter */
10
11    /* loop 20 times */
12    for ( i = 1; i <= 20; i++ ) {
13
14       /* pick random number from 1 to 6 and output it */
15       printf( "%10d", 1 + ( rand() % 6 ) );
16
17       /* if counter is divisible by 5, begin new line of output */
18       if ( i % 5 == 0 ) {
19          printf( "\n" );
20       } /* end if */
21
22    } /* end for */
23
24    return 0; /* indicates successful termination */
25
26 } /* end main */
```

Outline

fig05_07.c


```
     6         6         5         5         6
     5         1         1         5         3
     6         6         2         4         2
     6         2         3         4         1
```

Outline

Program Output

```c
/* Fig. 5.8: fig05_08.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main()
{
   int frequency1 = 0; /* rolled 1 counter */
   int frequency2 = 0; /* rolled 2 counter */
   int frequency3 = 0; /* rolled 3 counter */
   int frequency4 = 0; /* rolled 4 counter */
   int frequency5 = 0; /* rolled 5 counter */
   int frequency6 = 0; /* rolled 6 counter */

   int roll; /* roll counter */
   int face; /* represents one roll of the die, value 1 to 6 */

   /* loop 6000 times and summarize results */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6; /* random number from 1 to 6 */

```


fig05_08.c (Part 1 of 3)


```c
      /* determine face value and increment appropriate counter */
      switch ( face ) {

         case 1:          /* rolled 1 */
            ++frequency1;
            break;

         case 2:          /* rolled 2 */
            ++frequency2;
            break;

         case 3:          /* rolled 3 */
            ++frequency3;
            break;

         case 4:          /* rolled 4 */
            ++frequency4;
            break;

         case 5:          /* rolled 5 */
            ++frequency5;
            break;

```


fig05_08.c (Part 2 of 3)


```c

         case 6:          /* rolled 6 */
            ++frequency6;
            break;
      } /* end switch */

   } /* end for */

   /* display results in tabular format */
   printf( "%s%13s\n", "Face", "Frequency" );
   printf( "   1%13d\n", frequency1 );
   printf( "   2%13d\n", frequency2 );
   printf( "   3%13d\n", frequency3 );
   printf( "   4%13d\n", frequency4 );
   printf( "   5%13d\n", frequency5 );
   printf( "   6%13d\n", frequency6 );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```


fig05_08.c (Part 3 of 3)

Program Output


```c
/* Fig. 5.9: fig05_09.c
   Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

/* function main begins program execution */
int main()
{
   int i;         /* counter */
   unsigned seed; /* number used to seed random number generator */

   printf( "Enter seed: " );
   scanf( "%u", &seed );

   srand( seed ); /* seed random number generator */

   /* loop 10 times */
   for ( i = 1; i <= 10; i++ ) {

      /* pick a random number from 1 to 6 and output it */
      printf( "%10d", 1 + ( rand() % 6 ) );

```


fig05_09.c (Part 1 of 2)

```
23       /* if counter is divisible by 5, begin a new line of output */
24       if ( i % 5 == 0 ) {
25          printf( "\n" );
26       } /* end if */
27
28    } /* end for */
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
Enter seed: 867
         2         4         6         1         6
         1         1         3         6         2
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

Outline

fig05_09.c (Part 2 of 2)

Program Output

Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win

```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> /* contains prototype for function time */
6
7  /* enumeration constants represent game status */
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); /* function prototype */
11
12 /* function main begins program execution */
13 int main()
14 {
15    int sum;        /* sum of rolled dice */
16    int myPoint;    /* point earned */
17
18    enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */
19
20    /* randomize random number generator using current time */
21    srand( time( NULL ) );
22
23    sum = rollDice(  ); /* first roll of the dice */
24
```

Outline

fig05_10.c (Part 1 of 4)


```
25    /* determine game status based on sum of dice */
26    switch( sum ) {
27
28       /* win on first roll */
29       case 7:
30       case 11:
31          gameStatus = WON;
32          break;
33
34       /* lose on first roll */
35       case 2:
36       case 3:
37       case 12:
38          gameStatus = LOST;
39          break;
40
41       /* remember point */
42       default:
43          gameStatus = CONTINUE;
44          myPoint = sum;
45          printf( "Point is %d\n", myPoint );
46          break; /* optional */
47    } /* end switch */
48
```

Outline

fig05_10.c (Part 2 of 4)

```
49    /* while game not complete */
50    while ( gameStatus == CONTINUE ) {
51       sum = rollDice( ); /* roll dice again */
52
53       /* determine game status */
54       if ( sum == myPoint ) { /* win by making point */
55          gameStatus = WON;
56       } /* end if */
57       else {
58
59          if ( sum == 7 ) { /* lose by rolling 7 */
60             gameStatus = LOST;
61          } /* end if */
62
63       } /* end else */
64
65    } /* end while */
66
67    /* display won or lost message */
68    if ( gameStatus == WON ) {
69       printf( "Player wins\n" );
70    } /* end if */
71    else {
72       printf( "Player loses\n" );
73    } /* end else */
74
```

Outline

fig05_10.c (Part 3 of 4)


```
75    return 0; /* indicates successful termination */
76
77 } /* end main */
78
79 /* roll dice, calculate sum and display results */
80 int rollDice( void )
81 {
82    int die1;    /* first die */
83    int die2;    /* second die */
84    int workSum; /* sum of dice */
85
86    die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
87    die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
88    workSum = die1 + die2;     /* sum die1 and die2 */
89
90    /* display results of this roll */
91    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
92
93    return workSum; /* return sum of dice */
94
95 } /* end function rollRice */
```

Outline

fig05_10.c (Part 4 of 4)


```
Player rolled 5 + 6 = 11
Player wins

Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins

Player rolled 1 + 1 = 2
Player loses

Player rolled 1 + 4 = 5
Point is 5
Player rolled 3 + 4 = 7
Player loses
```

Outline

Program Output

Storage Classes

- Storage class specifiers
  - Specificano la durata di un oggetto in memoria
- Automatic storage
  - Gli oggetti sono creati e distrutti all'interno del loro blocco
  - `auto`: default for variabili locali

    ```
    auto double x, y;
    ```
  - `register`: tenta di inserire la variabile in registri ad alta velocità

    ```
    register int counter = 1;
    ```

Storage Classes

- Static storage
  - Le variabili esistono per l'intera esecuzione del programma
  - Il valore di default è zero
  - `static`: variabili locali definite nelle funzioni.
    - Mantengono il valore dopo il termine della funzione
    - Conosciute solo nella propria funzione
  - `extern`: default per variabili globali e funzioni
    - Conosciute in ogni funzione

Scope Rules

- File scope
  - Gli identificatori definiti fuori dalla funzione sono conosciuti in tutte le funzioni
  - Usato per le variabili globali, la definizione di funzioni, e i prototipi di funzione
- Function scope
  - Possono essere referenziati solo all'interno del corpo di una funzione

Scope Rules

- Block scope
  - Identificatori dichiarati all'interno di un blocco
    - La visibilità nel blocco comincia dalla definizione e termina con la parentesi destra di chiusura del blocco
  - Usato per variabili, parametri di funzioni
  - Blocchi esterni "nascosti" dai blocchi interni se esiste una variabile con lo stesso nome nel blocco interno
- Function prototype scope
  - Usato per gli identificatori nella lista dei parametri

```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void );       /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void );      /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main()
13 {
14    int x = 5; /* local variable to main */
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { /* start new scope */
19       int x = 7; /* local variable to new scope */
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } /* end new scope */
23
24    printf( "local x in outer scope of main is %d\n", x );
25
```

Outline

fig05_12.c (Part 1 of 3)

```
26    useLocal();       /* useLocal has automatic local x */
27    useStaticLocal(); /* useStaticLocal has static local x */
28    useGlobal();      /* useGlobal uses global x */
29    useLocal();       /* useLocal reinitializes automatic local x */
30    useStaticLocal(); /* static local x retains its prior value */
31    useGlobal();      /* global x also retains its value */
32
33    printf( "local x in main is %d\n", x );
34
35    return 0; /* indicates successful termination */
36
37 } /* end main */
38
39 /* useLocal reinitializes local variable x during each call */
40 void useLocal( void )
41 {
42    int x = 25;  /* initialized each time useLocal is called */
43
44    printf( "\nlocal x in a is %d after entering a\n", x );
45    x++;
46    printf( "local x in a is %d before exiting a\n", x );
47 } /* end function useLocal */
48
```

Outline

g05_12.c (Part 2
f 3)


```
49 /* useStaticLocal initializes static local variable x only the first time
50    the function is called; value of x is saved between calls to this
51    function */
52 void useStaticLocal( void )
53 {
54    /* initialized only first time useStaticLocal is called */
55    static int x = 50;
56
57    printf( "\nlocal static x is %d on entering b\n", x );
58    x++;
59    printf( "local static x is %d on exiting b\n", x );
60 } /* end function useStaticLocal */
61
62 /* function useGlobal modifies global variable x during each call */
63 void useGlobal( void )
64 {
65    printf( "\nglobal x is %d on entering c\n", x );
66    x *= 10;
67    printf( "global x is %d on exiting c\n", x );
68 } /* end function useGlobal */
```

Outline

fig05_12.c (Part 3
of 3)


```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Outline

Program Output

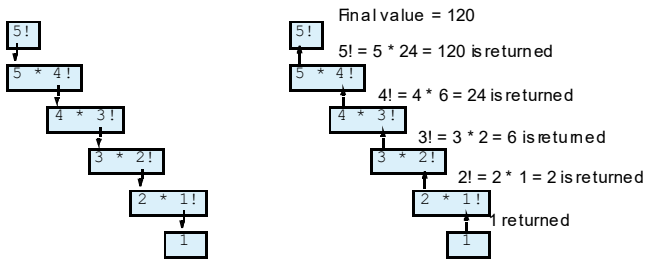
Ricorsione

- Funzioni ricorsive
  - Funzioni che richiamano se stesse
  - Necessità di un livello assiomatico
  - La funzione lancia una copia di se stessa (passo ricorsivo) per risolvere il problema

Ricorsione

- Esempio: fattoriale
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Nota che
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Possibile computazione ricorsiva
  - Livello assiomatico (caso base) (`1! = 0! = 1`)
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

Ricorsione



(a) Sequence of recursive calls. (b) Values returned from each recursive call.

```
1  /* Fig. 5.14: fig05_14.c
2     Recursive factorial function */
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    int i; /* counter */
11
12    /* loop 10 times.  During each iteration, calculate
13       factorial( i ) and display result */
14    for ( i = 1; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* end for */
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

Outline

fig05_14.c (Part 1 of 2)


```
22 /* recursive definition of function factorial */
23 long factorial( long number )
24 {
25    /* base case */
26    if ( number <= 1 ) {
27       return 1;
28    } /* end if */
29    else { /* recursive step */
30       return ( number * factorial( number - 1 ) );
31    } /* end else */
32
33 } /* end function factorial */
```

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Outline

fig05_14.c (Part 2 of 2)

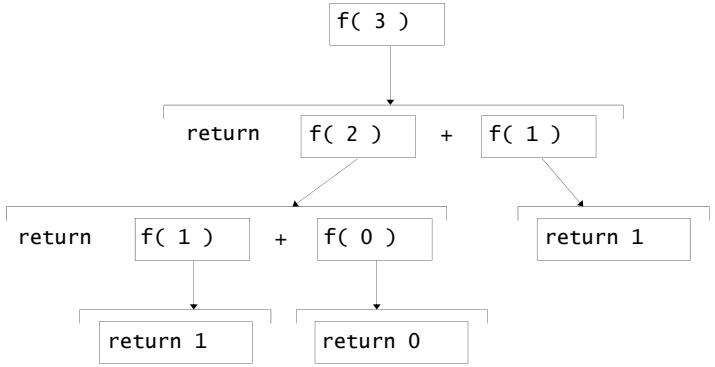
La serie di Fibonacci

- Serie di Fibonacci: 0, 1, 1, 2, 3, 5, 8...
  - Ogni numero è la somma dei due precedenti
  - Definizione ricorsiva:
    - `fib( n ) = fib( n - 1 ) + fib( n - 2 )`
  - Codice per la funzione di `fibonacci`

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
    return n;
  else
    return fibonacci( n - 1) +
      fibonacci( n - 2 );
}
```


La serie di Fibonacci

- Set di chiamate ricorsive alla funzione `fibonacci`




```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    long result; /* fibonacci value */
11    long number; /* number input by user */
12
13    /* obtain integer from user */
14    printf( "Enter an integer: " );
15    scanf( "%ld", &number );
16
17    /* calculate fibonacci value for number input by user */
18    result = fibonacci( number );
19
20    /* display result */
21    printf( "Fibonacci( %ld ) = %ld\n", number, result );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
26
```

Outline

fig05_15.c (Part 1 of 2)


```
27 /* Recursive definition of function fibonacci */
28 long fibonacci( long n )
29 {
30    /* base case */
31    if ( n == 0 || n == 1 ) {
32       return n;
33    } /* end if */
34    else { /* recursive step */
35       return fibonacci( n - 1 ) + fibonacci( n - 2 );
36    } /* end else */
37
38 } /* end function fibonacci */
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0

Enter an integer: 1
Fibonacci( 1 ) = 1

Enter an integer: 2
Fibonacci( 2 ) = 1

Enter an integer: 3
Fibonacci( 3 ) = 2

Enter an integer: 4
Fibonacci( 4 ) = 3
```

Outline

fig05_15.c (Part 2 of 2)

Program Output

```
Enter an integer: 5
Fibonacci( 5 ) = 5

Enter an integer: 6
Fibonacci( 6 ) = 8

Enter an integer: 10
Fibonacci( 10 ) = 55

Enter an integer: 20
Fibonacci( 20 ) = 6765

Enter an integer: 30
Fibonacci( 30 ) = 832040

Enter an integer: 35
Fibonacci( 35 ) = 9227465
```

Outline

Program Output (continued)

Ricorsione vs. Iterazione

- Ripetizione
  - Iterazione: loop esplicito
  - Ricorsione: chiamate ripetute di funzione
- Terminazione
  - Iterazione: fallisce la condizione del loop
  - Ricorsione: passo base (livello assiomatico) riconosciuto
- Possibilità di loop infiniti per entrambe
- Balance
  - Scelta tra performance (iterazione) e buon software engineering (ricorsione)