Funzioni

Dichiarazione di funzioni
Definizione di funzioni
Chiamata di funzioni
Passaggio per valore

I sottoprogrammi

- Un sottoprogramma (o subroutine, o funzione) è uno strumento per strutturare i programmi, sia per renderli più facili da capire che per permettere il riutilizzo del codice
- Le procedure consentono al programmatore di concentrarsi su una parte del problema: i parametri rappresentano una barriera fra la procedura ed il resto del programma e permettono lo scambio dei dati
  - Permettono di modularizzare i programmi
  - Variabili definite nei sottoprogrammi sono visibili e utilizzabili solo nei sottoprogrammi
- Vantaggi:
  - Evitare la duplicazione del codice
  - Creare programmi strutturati per raffinamenti successivi (e quindi più leggibili)
  - Divide et impera: lo sviluppo dei programmi è più semplice
  - Riusabilità del Software
    - Metodi esistenti possono servire da mattoni per nuovi programmi
    - Astrazione - I dettagli di implementazione vengono nascosti (funzioni di libreria)

I sottoprogrammi

- Un sottoprogramma
  - è una sequenza di istruzioni che vengono attivate a seguito di una chiamata

- Problemi connessi con i sottoprogrammi
  - Determinazione dell'indirizzo di rientro al codice chiamante
  - Scambio di informazioni fra sottoprogramma e codice chiamante
  - Passaggio dei parametri in ingresso e in uscita

Le funzioni in C

- Una funzione è una porzione di codice che può essere richiamata in un programma
- Come le funzioni matematiche, ogni funzione restituisce un valore e riceve una serie di parametri (anche 0)
- il tipo del valore di ritorno può essere primitivo o derivato, oppure il tipo void, cioè nulla

Le funzioni

- L'uso delle funzioni aumenta chiarezza, modularità e riutilizzabilità di un programma
- Sono utili per svolgere più volte la stessa sequenza di operazioni
- L'esecuzione può essere condizionata dal valore di opportuni parametri

Le funzioni

- Le funzioni possono essere definite (una volta sola) nel programma
- Possono essere reperite in librerie di codice di utilità generale, ovvero collezioni di funzioni implementate per la risoluzione di problemi specifici
- esempi:

  `printf, atoi e atof`

- Sono state scritte una sola volta
- Le istruzioni che le compongono sono richiamate ogni volta che si chiama la funzione

Dichiarazione di funzioni

- serve per segnalare al compilatore l'esistenza di una determinata funzione
- viene specificato il prototipo:
  - il nome
  - il tipo ritornato
  - l'elenco degli argomenti (o parametri)
- non prevede che si specifichino le istruzioni
- è opzionale

Esempio di dichiarazione

- il seguente codice:

```
int secondi(int h, int m, int s);
```

- dichiara la funzione secondi, che:
  - restituisce un valore di tipo int
  - richiede tre argomenti sempre di tipo int

- in una dichiarazione non è obbligatorio specificare il nome degli argomenti della funzione, ma solo il tipo

Esempio di dichiarazione

- È altrettanto valido il codice:

```
int secondi(int, int, int);
```

- dove compaiono i nomi degli argomenti
  - è preferibile però assegnare un nome agli argomenti
  - aumenta la leggibilità del codice
  - facilita l'uso della funzione

Dichiarazione di funzioni

- con la dichiarazione non si specificano le istruzioni che compongono la funzione
- la dichiarazione informa il compilatore che la funzione, comprensiva delle relative istruzioni, è definita altrove
- grazie alla dichiarazione, quando il compilatore incontra una chiamata a funzione, può effettuare tutti i controlli necessari sulla correttezza della chiamata
- la dichiarazione di una funzione deve sempre precedere nel sorgente la prima invocazione della stessa
- la definizione, invece, può essere presente in un qualunque punto del sorgente o in una libreria esterna

Uso di void

- Le funzioni possono restituire void come valore di ritorno
- Ad esempio:

```
void exit(int status);
```

- termina immediatamente il programma
  - restituisce al sistema il valore intero status
  - Ritorna void (cioè nulla) al programma

Uso di void

- Le funzioni possono accettare il tipo void come argomento
- Ad esempio:

```
int rand(void);
```

- restituisce un valore pseudo-casuale intero
- non richiede alcun parametro, ovvero accetta void

Definizione di funzioni

Una definizione è costituita da due parti:

- una dichiarazione, nella quale, come visto, sono elencati: il tipo di dato restituito dalla funzione, il nome della funzione, il tipo e nome degli eventuali argomenti
- il corpo della funzione, racchiuso tra parentesi graffe e comprendente zero o più di queste componenti:
  - dichiarazioni e definizioni di variabili locali
  - istruzioni
  - istruzioni return

```
1.  /* esempio di definizione */
2.  secondi(int h, int m, int s)
3.  {
4.    return (3600 * h + 60 * m + s);
5.  }
```

Definizione di funzioni

- tutte le parti che compongono il corpo di una funzione sono opzionali
  - infatti, la seguente funzione è lecita:

```
void do_nothing(void) { }
```

- il corpo di do_nothing non comprende alcuna istruzione

Chiamata di funzioni

- La chiamata di una funzione, o invocazione, è l'operazione con la quale si richiama l'esecuzione della funzione stessa

```
int sec = secondi(1, 15, 0);
```

- Alla funzione vengono passati i 3 parametri 1, 15 e 0, e il risultato restituito dalla funzione viene assegnato alla variabile sec

- Quando si incontra una funzione il controllo viene trasferito alla prima istruzione della funzione stessa

Chiamata di funzioni

- Possono comparire zero o più chiamate ad una funzione
- Per richiamare una funzione si deve utilizzare il nome della funzione seguita dagli argomenti passati alla funzione racchiusi da parentesi tonde e separati da virgole
- Ogni chiamata deve passare sempre lo stesso numero e tipo di argomenti
- Una funzione termina l'esecuzione quando
  - si incontra l'istruzione return, oppure
  - si esegue la sua ultima istruzione

Funzioni matematiche

- Numerose funzione matematiche sono a disposizione dei programmatori
- È necessario includere il file delle definizioni con l'istruzione

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

- Alcuni compilatori richiedono che si indichi esplicitamente l'uso della libreria matematica

```
gcc -Wall sqrt.c -lm
```

Funzioni matematiche

Il tipo restituito è un intero

`abs(x)` **Restituisce il valore assoluto**

Il tipo restituito è un double

`fabs(x)` **Restituisce il valore assoluto**

`ceil(x)` **Restituisce l'intero più piccolo maggiore o uguale all'argomento**

`floor(x)` **Restituisce l'intero più grande minore o uguale all'argomento**

`round(x)` **Restituisce l'intero più vicino a x**

`sqrt(x)` **Restituisce la radice quadrata**

`pow(a, b)` **Operazione di elevamento a potenza a^b**

`exp(x)` **Funzione esponenziale**

`log(x)` **Restituisce il logaritmo naturale (in base e)**

Funzioni trigonometriche

`sin(x)` **Restituisce il seno dell'angolo**
`cos(x)` **Restituisce il coseno dell'angolo**
`tan(x)` **Restituisce la tangente dell'angolo**
`acos(x)` **Funzione inversa di cos**
`asin(x)` **Funzione inversa di sin**
`atan(x)` **Funzione inversa di tan**
`atan2(y, x)` **Restituisce la componente θ del punto (r, θ) in coordinate polari che corrisponde a (x, y) in coordinate Cartesiane**
per x positivo `atan2(y,x)==atan(y/x)`

Per riferirsi a π si può utilizzare la costante M_PI

Fanno sempre riferimento a valori di angoli in radianti non in gradi
Eventualmente si usi la conversione

```
radianti = gradi*M_PI/180
```

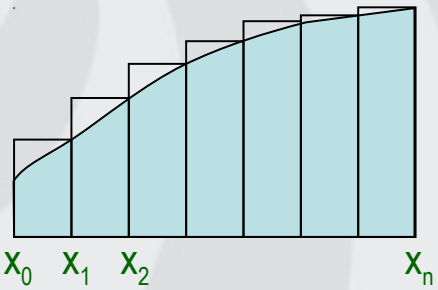
FunzioniTrigonometriche.java

```
1.  #include <stdio.h>
2.  #include <math.h>
3.
4.  int main(int argc, char **argv)
5.  {
6.     int gradi = 45;
7.     double radianti = gradi * M_PI / 180;
8.     double seno = sin(radianti);
9.     double pi = asin(seno)*4;
10.    printf("sen(45°)=%.16f\n", seno);
11.    printf("pi greco=%.16f\n", pi);
12.    printf("M_PI=    %.16f\n", M_PI);
13. }
```

```
sen(45°)=0.7071067811865475
pi greco=3.1415926535897927
M_PI=    3.1415926535897931
```

Calcolo dell'integrale

Un integrale può essere solo approssimato!



$$I = f(x_1)(x_1-x_0)+ f(x_2)(x_2-x_1)+\dots +f(x_n)(x_n-x_{n-1})$$

Nell'ipotesi che tutti i punti siano equispaziati

$$I = (f(x_1) + f(x_2) +\dots + f(x_n))\Delta x$$

$$\Delta x =(x_n-x_0)/n$$

$$x_i=f(x_0+\Delta x \times i)$$

L'errore che si compie diminuisce al crescere di n

Calcolo dell'integrale

```
1.  int n = 100, i;
2.  double a = 0; // valore iniziale
3.  double b = M_PI_2;  // π/2 valore finale
4.  double somma = 0, delta = (b-a)/n;
5.  for(i=1; i<=n; i++) {
6.      double x = i * delta;
7.      somma += sin(x);
8.  }
9.  somma *= delta;
10. printf("Integrale: %f\n", somma);
```

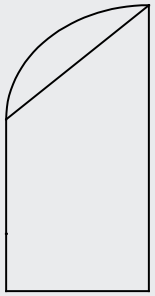
```
Integrale: 1.007833
```

```
Integrale: 1.000079
con n=10000
```

Calcolo dell'integrale

Ovviamente ci possono essere anche approcci diversi:

- $I = (f(x_0) + f(x_1) + \ldots + f(x_{n-1}))\ \Delta x$
- $I = (f(x_0+0.5) + f(x_1+0.5) + \ldots + f(x_{n-1}+0.5))\ \Delta x$
- Approssimazione tramite trapezi (invece di rettangoli)



Passaggio dei parametri

- Passaggio dei parametri in ingresso e in uscita
  - Parametri formali: simboli che rappresentano i dati su cui il sottoprogramma opera (specificati nella definizione del sottoprogramma stesso)
  - Parametri attuali: i dati corrispondenti ai parametri formali, su cui il sottoprogramma effettivamente opera
- Una funzione utilizza i parametri per svolgere il suo lavoro, la comunicazione fra il codice chiamante ed il sottoprogramma avviene sempre secondo la modalità del passaggio per valore

Passaggio dei parametri

```
1.      int esempio(int a)
2.      {
3.          int ris;
4.          ris = a*2;
5.          return ris;
6.      }
7.
8.      int main(int argc, char *argv[])
9.      {
10.         int ris = esempio(1);
11.         printf("%d\n", ris);
12.         return 0;
13.     }
14.
```

Definizione di una funzione

Parametri formali

Definizione del valore restituito

Attivazione di una funzione

Parametri attuali

Utilizzo del valore restituito

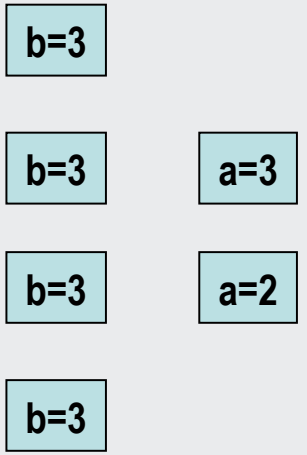
Si noti che esistono due variabili ris, completamente indipendenti fra di loro

Il passaggio per valore

- Secondo la modalità del passaggio per valore ogni funzione ha una propria zona di memoria per memorizzare i dati (messa a disposizione solo al momento dell'effettivo utilizzo e rilasciata quando non è più necessaria), al momento dell'uso della funzione i parametri sono copiati, quindi non vi è un accesso diretto ai valori del codice chiamante

```
1.  void sottoprogramma(int a)
2.  {
3.       //… …
4.       a = 2;
5.  }
6.
7.  // … …
8.  int b=3;
9.  sottoprogramma(b);
10. printf("b=%d\n", b);
11. // … …
```

b=3



Il passaggio per valore

```
1.  void scambia(int a, int b) {
2.      int tmp=a;
3.      a=b;
4.      b=tmp;
5.  }
6.
7.  // … …
8.  int a=2, b=3;
9.  scambia(b, a);
10. printf("a=%d b=%d\n", a, b);
11. // … …
```

Non vi è corrispondenza fra i nomi di metodi diversi

```
a=2 b=3
```

Non ho ottenuto il risultato voluto a causa del passaggio per valore

Il passaggio per valore

a=2	b=3

Chiamata scambio – viene allocata memoria per le variabili locali

a=2	b=3	a=3	b=2	tmp=?	Copia parametri
		a=3	b=2	tmp=3	int tmp=a;
		a=2	b=2	tmp=3	a=b;
		a=2	b=3	tmp=3	b=tmp;

Ritorno da scambio – viene liberata la memoria per le variabili locali

a=2	b=3

Scambio

```
1.  // attenzione:        a=2    b=3
2.       a=b;            a=3    b=3
3.       b=a;            a=3    b=3
4.  // è un modo scorretto di fare lo scambio fra variabili
5.  // perdo il valore di a
```

Scambio tramite exor

```
1.  //                          a=A          b=B
2.  a = a^b;                    a=A^B        b=B
3.  b = a^b;                    a=A^B        b=A^B^B
4.  a = a^b;                    a=A^B^A      b=A
5.  // faccio a meno della variabile temporanea


6.  // l'exor è commutativo e associativo
7.  // inoltre ∀X X^X → 0 e X^0 → X
```

L'uso di puntatori

```
1.  void scambia(int *a, int *b)
2.  {
3.      int tmp = *a;
4.      *a = *b;
5.      *b = tmp;
6.  }
7.
8.  // … …
9.  a=2, b=3;
10. scambia(&b, &a);
11. printf("a=%d b=%d\n", a, b);
12. // … …
```

Tramite i puntatori accedo allo stesso dato *in memoria,* in questo caso si è copiato l'indirizzo di memoria

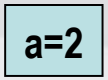
```
a=3 b=2
```

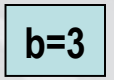
Tramite l'uso dei puntatori posso simulare il passaggio per indirizzo (viene copiato non il contenuto di una cella di memoria, ma il suo indirizzo

sottoprogramma e programma lavorano quindi con gli stessi dati)

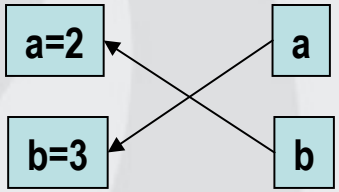
Nel passaggio per indirizzo chiamante e sottoprogramma condividono un indirizzo di memoria

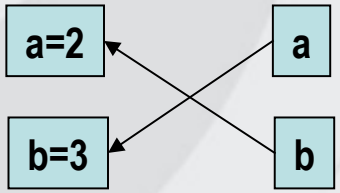
L'uso di puntatori



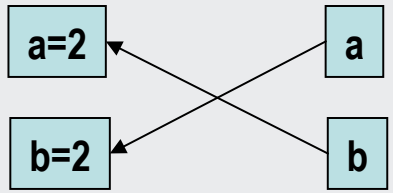


Chiamata scambio

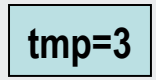


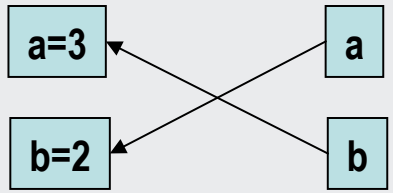


tmp=*a

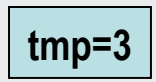


*a=*b

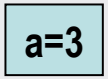


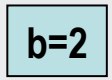


*b=tmp



Ritorno da scambio



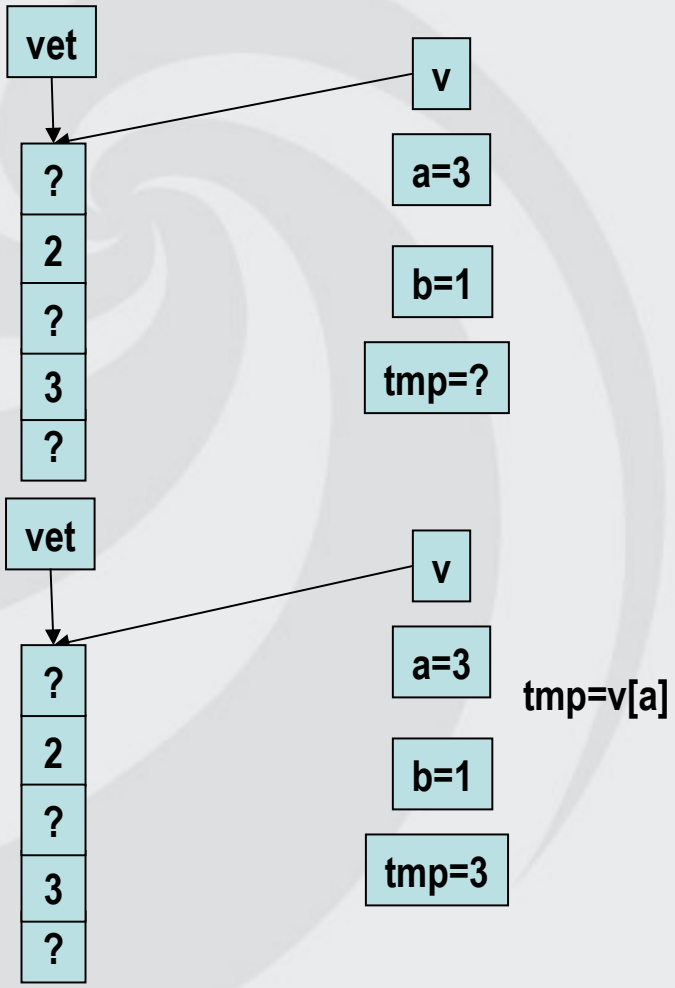


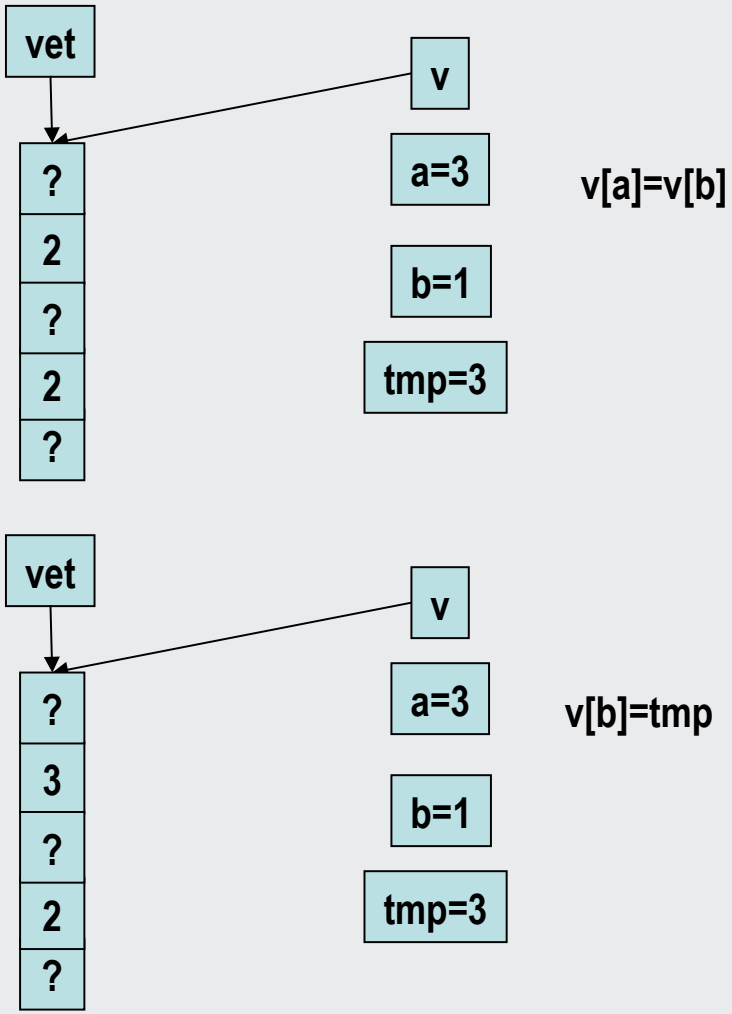
Array e indici

```
1.  void scambia(int v[], int a, int b) {
2.      int tmp = v[a];
3.      v[a] = v[b];
4.      v[b] = tmp;
5.  }
6.
7.  // … …
8.  int vet[5];
9.  vet[1] = 2;
10. vet[3] = 3;
11. scambia(vet, 3, 1);
12. printf("v[1]=%d vet[3]=%d\n", v[1], v[3]);
13. // … …
```

```
V[1]=3 v[3]=2
```

Array e indici





Passaggio per indirizzo

- Il passaggio per indirizzo (ovvero l'uso di puntatori) permette ad una funzione di valutare più di un valore e di restituirli al chiamante
  - Il comportamento è comunque di più difficile interpretazione
  - Conviene commentare opportunamente il codice
- L'uso dei puntatori è frequente quando si vuole evitare di copiare grandi quantità di dati ogni volta che viene attivata una funzione