

# Funzioni

Corso di Programmazione di Sistema

Dr. Nicola Biccocchi

DIEF/UNIMORE

Marzo 2022

- Un funzione è una sequenza di istruzioni che vengono attivate a seguito di una apposita chiamata
- Vantaggi:
  - favoriscono modularizzazione del codice
  - favoriscono il riuso del codice (librerie)
  - favoriscono lo sviluppo incrementale (creazione di interfacce che disaccoppiano parti di software)
  - favoriscono la leggibilità del codice
- Svantaggi:
  - Determinazione dell'indirizzo di rientro al codice chiamante
  - Scambio di informazioni fra funzioni e codice chiamante (passaggio di parametri)

# Dichiarazione di funzioni

- Serve per segnalare al compilatore l'esistenza di una determinata funzione (e come invocarla) ma non specifica le istruzioni che compongono la funzione
- 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
- La dichiarazione specifica il *prototipo* della funzione:
  - il tipo ritornato
  - il nome della funzione
  - l'elenco dei parametri (argomenti)
- In fase di dichiarazione è consentito omettere il nome dei parametri

```
1  int secondi(int h, int m, int s);  
2  /* oppure */  
3  int secondi(int, int, int);
```

# Definizione di funzioni

Una definizione è costituita da due parti:

- la *dichiarazione della funzione*
- il *corpo della funzione*, racchiuso tra parentesi graffe e comprendente zero o più di queste componenti:
  - dichiarazioni e definizioni di variabili
  - istruzioni
  - istruzione return

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

# Invocazione di funzioni

- L'*invocazione* di una funzione è l'operazione con la quale si richiama l'esecuzione della funzione
- Per richiamare una funzione si deve utilizzare il nome della funzione seguita dagli argomenti racchiusi da parentesi tonde e separati da virgole
- Un'invocazione di funzione trasferisce il controllo alla prima istruzione della funzione stessa
- Una funzione termina quando: (a) viene eseguita l'istruzione *return*, oppure (b) viene eseguita l'ultima istruzione

```
1  int secondi(int h, int m, int s) {  
2      return (3600 * h + 60 * m + s);  
3  }  
4  
5  int main() {  
6      int h=1, m=1, s=1, totale_secondi;  
7      totale_secondi = secondi(h,m,s);  
8      printf("Totale secondi: %d\n", totale_secondi);  
9  }
```

# Tipo void

- L'uso del tipo *void* nelle funzioni identifica *tipi nulli*
- Se usato come tipo di ritorno, la funzione non restituisce alcun valore
- Se usato come parametro di input, la funzione non accetta nessun parametro

```
1 void say_hi(void) {  
2     printf("Hi!\n");  
3 }  
4  
5 int main() {  
6     say_hi();  
7     return 0;  
8 }
```

# Parametri di input

- È obbligatorio indicare il tipo delle variabili. Se non ci sono variabili, si usa il tipo speciale *void*
- Non è possibile indicare parametri facoltativi
- È possibile indicare funzioni con numero di parametri variabile, (vedi *printf*)
- Il passaggio dei parametri avviene *sempre per copia (per valore)*

```
1 void try_modification(int value) {  
2     // solo la copia ricevuta dalla funzione viene modificata  
3     value = 0;  
4 }  
5  
6 int main(){  
7     int a=5;  
8     try_modification(a);  
9     printf("a = %d\n", a);  
10    return 0;  
11 }
```

# Visibilità e tempo di vita delle variabili locali

- Le variabili che abbiamo utilizzato fin'ora sono *variabili locali*, visibili solo all'interno della funzione
- Le funzioni invocate *non hanno accesso* alle variabili di livello superiore!
- La gestione della memoria delle variabili locali è *automatica*
  - Le variabili vengono allocate al momento dell'invocazione della funzione
  - Le variabili vengono de-allocate al momento del ritorno della funzione
- Ad ogni invocazione, le variabili della funzione non dipendono dalle esecuzioni precedenti!



# Variabili globali

- Il C supporta anche *variabili globali*, visibili sempre e da tutte le funzioni
- *Preferibile limitare utilizzo di variabili globali, oppure utilizzarle come costanti*

```
1  #include<stdio.h>
2
3  int a;
4  /* const int a; -> errore! */
5
6  void try_modification(void){
7      a = 10;
8  }
9
10 int main(){
11     a=5;
12     try_modification();
13     printf("a=%d\n", a);
14 }
```

# Variabili locali static

- Una variabile locale è detta *static* se il suo tempo di vita corrisponde a quello del processo
- E' possibile utilizzare variabili static per avere funzioni che mantengono uno stato fra diverse invocazioni

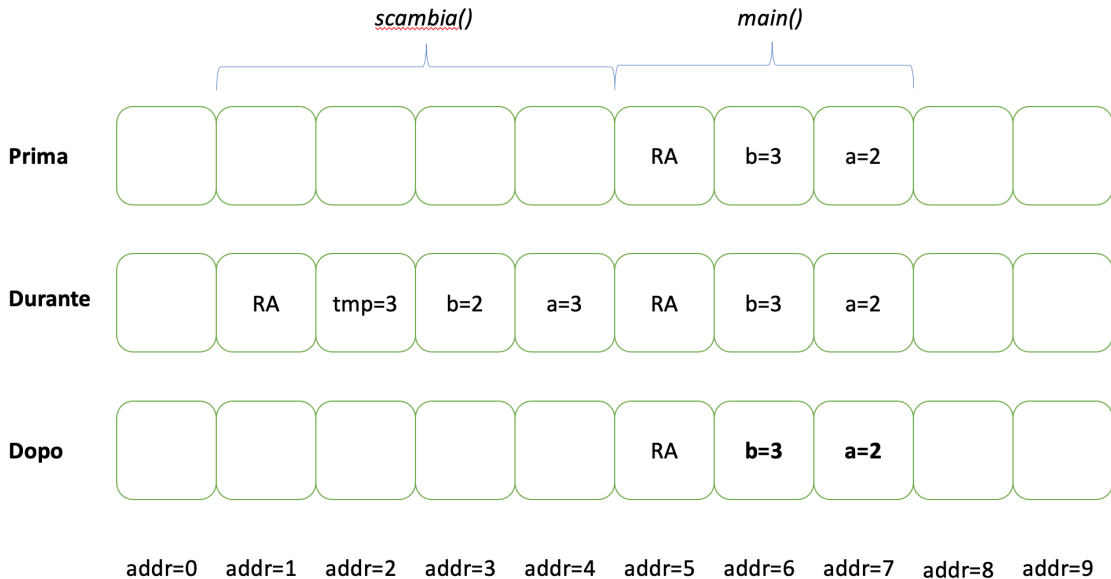
```
1 void counter() {  
2     static int count=0;  
3  
4     count++;  
5     printf("count=%d\n", count);  
6 }  
7  
8 int main(){  
9     counter();  
10    counter();  
11  
12    return 0;  
13 }
```

# Passaggio per valore (copia)

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

# Passaggio per valore (copia)

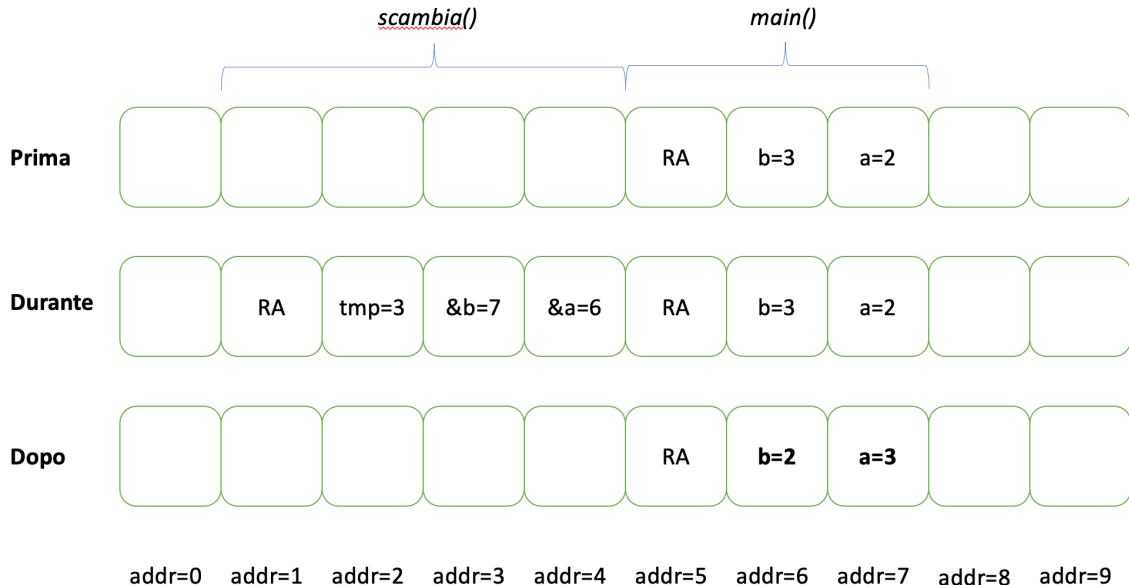


# Passaggio per riferimento (copia del riferimento)

- Permette alla funzione chiamata di modificare il valore della variabile passata dal chiamante
- Evita la copia di variabili voluminose
- Contente alla funzione chiamata di ritornare più di un valore di ritorno
- Il passaggio per riferimento *implica il passaggio per valore di un puntatore alla variabile*

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

# Passaggio per riferimento (copia del riferimento)



# Passaggio di puntatori const

- Talvolta è necessario passare alla funzione variabili di grandi dimensioni (array, matrici)
- Per evitare la copia della variabile si usa un puntatore alla variabile. Tuttavia, questa possibilità si scontra con il fatto che, tramite il puntatore, la funzione può modificare i dati del chiamante
- Questo può determinare che un errore di implementazione nella funzione propaghi esiti non voluti. Il problema è risolvibile attraverso *puntatori definiti costanti*

```
1 void scambia(const int *a, const int *b) {  
2     int tmp = *a;  
3     *a = *b;           // errori di compilazione!  
4     *b = tmp;          // errori di compilazione!  
5 }  
6  
7 int main() {  
8     int a = 2, b = 3;  
9     scambia(&b, &a);  
10    printf("a=%d b=%d\n", a, b);  
11 }
```

# Vettori di puntatori (a carattere)

- Gli elementi di un vettore possono essere di qualunque tipo:
  - numeri interi, virgola mobile, caratteri, ma anche *puntatori*
  - ad esempio `char *settimana[]` è un vettore che memorizza 7 puntatori a carattere
  - tipo in genere utilizzato per gestire gruppi di stringhe di caratteri

```
1 char *settimana[] = {
2     "lunedì",
3     "martedì",
4     "mercoledì",
5     "giovedì",
6     "venerdì",
7     "sabato",
8     "domenica"
9 };
10
11 printf("%s\n", settimana[0]); /* Output: lunedì */
```



# Vettori di puntatori (a carattere)

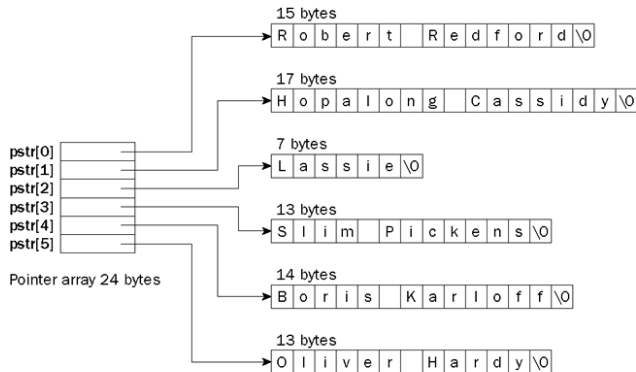


Figura 3: Puntatore a puntatore a carattere

# Passaggio di parametri al programma principale

- E' possibile passare parametri *dalla shell ad un programma C* utilizzando parametri opzionali della funzione `main()`
- **argc** è un numero intero e rappresenta il numero dei parametri ricevuti (considerando anche il comando stesso)
- **argv** è un vettore di stringhe che rappresenta i parametri stessi (`argv[0]` è il comando stesso)

```
1  int main(void) {  
2  
3  }
```

```
1  int main(int argc, char *argv[]) {  
2  
3  }
```

# Passaggio di parametri al programma principale

```
1  int main(int argc, char *argv[]) {  
2      int i;  
3  
4      for (i=0; i<argc; i++) {  
5          printf("[%d] %s\n", i, argv[i]);  
6      }  
7  }
```

```
1  $ ./a.out ciao nicola bicocchi  
2  [0] ./a.out  
3  [1] ciao  
4  [2] nicola  
5  [3] bicocchi
```

# Passaggio di parametri al programma principale

```
1  int main(int argc, char *argv[]) {  
2      int a;  
3      double b;  
4      char c[128];  
5  
6      if (argc != 4) {  
7          printf("usage: %s int double char[]\n", argv[0]);  
8          exit(1);  
9      }  
10  
11     a = atoi(argv[1]);  
12     b = atof(argv[2]);  
13     strncpy(c, argv[3], sizeof(c));  
14     printf("%d %f %s\n", a, b, c);  
15 }
```

- Una funzione è definita in modo ricorsivo se è definita in termini di se stessa.
- Nella definizione ricorsiva di una funzione è possibile identificare *casi base* e *casi ricorsivi*:
- I casi base permettono di calcolare il valore della funzione, anche se solo nei casi più semplici
- I casi ricorsivi permettono di calcolare la funzione mediante altre valutazioni della funzione

```
1  n! = 1 x 2 x ... x (n-2) x (n-1) x n
2
3  n! = 1                (if n == 0) // caso base
4  n! = n x (n-1)!      (if n > 0)  // caso ricorsivo
```

# Ricorsione (fattoriale)

```
1 // implementazione fattoriale iterativa
2 int fatt(int n) {
3     int fatt, i;
4     for (i=1, fatt=1; i<=n; i++)
5         fatt = fatt * i;
6     return fatt;
7 }
```

```
1 // implementazione fattoriale ricorsiva
2 int fatt_r(int n) {
3     if (n == 0)
4         return 1;
5     return n * fatt_r(n - 1);
6 }
```

# Ricorsione (fibonacci)

```
1 // implementazione fibonacci iterativa
2 int fibonacci(int n) {
3     int x = 0, y = 1, z = 0;
4     for (int i = 0; i < n; i++) {
5         z = x + y;
6         x = y;
7         y = z;
8     }
9     return z;
10 }
```

```
1 // implementazione fibonacci ricorsiva
2 int fibonacci_r(int n) {
3     if (n == 0) return 0;
4     if (n == 1) return 1;
5     return (fibonacci(n-1) + fibonacci(n-2));
6 }
```