

Laboratorio di Informatica

Programmazione Modulare

(Parte 1)

docente: Cataldo Musto

cataldo.musto@uniba.it

Programmazione Modulare

Introduzione

Programmazione Modulare

- Cosa è?

Programmazione Modulare

- Cosa è?
 - I moderni Linguaggi di Programmazione offrono la possibilità di suddividere i programmi **in sotto-programmi, frammenti di codice più piccoli, distinti dal programma principale (main())**
 - Questi frammenti di codice prendono il nome **di funzioni (o di procedure)**
 - Ogni Linguaggio di Programmazione definisce dei meccanismi per **definire** dei sotto-programmi e dei meccanismi per **invocarli**.

Programmazione Modulare

- Cosa è?
 - I moderni Linguaggi di Programmazione offrono la possibilità di suddividere i programmi **in sotto-programmi, frammenti di codice più piccoli, distinti dal programma principale (main())**
 - **Questi frammenti di codice prendono il nome di funzioni (o di procedure)**
 - Ogni Linguaggio di Programmazione definisce dei meccanismi per **definire** dei sotto-programmi e dei meccanismi per **invocarli**.
 - **Funzioni e Procedure sono un meccanismo di astrazione**

Astrazione?

Programmazione Modulare

Astrazione?

Astrazione - in filosofia è un metodo logico per ottenere concetti universali ricavandoli dalla conoscenza sensibile di oggetti particolari mettendo da parte ogni loro caratteristica spazio-temporale. **Astrazione** - nell'arte il termine designa la creazione di un segno astratto.

[Astrazione - Wikipedia](https://it.wikipedia.org/wiki/Astrazione)

<https://it.wikipedia.org/wiki/Astrazione>

Programmazione Modulare

Astrazione?

Astrazione - in filosofia è un metodo logico per ottenere concetti universali ricavandoli dalla conoscenza sensibile di oggetti particolari mettendo da parte ogni loro caratteristica spazio-temporale. **Astrazione** - nell'arte il termine designa la creazione di un segno astratto.

[Astrazione - Wikipedia](https://it.wikipedia.org/wiki/Astrazione)

<https://it.wikipedia.org/wiki/Astrazione>

Programmazione Modulare

- **Funzioni e Procedure sono un meccanismo di astrazione**
 - **Le funzioni sono un esempio di astrazione sui dati**, perchè ci permettono di estendere gli **operatori** disponibili nel linguaggio
 - **Le procedure sono un esempio di astrazione sulle istruzioni**, perchè ci permettono di estendere **le istruzioni primitive** disponibili nel linguaggio
- **Torneremo su questi concetti.**

Programmazione Modulare

Paradigma Divide-et-Impera

La programmazione modulare permette di implementare il **paradigma 'divide-et-impera'**, che aiuta a gestire meglio la complessità dei problemi, suddividendoli in problemi più piccoli di dimensioni ridotte.

Programmazione Modulare

Paradigma Divide-et-Impera

La programmazione modulare permette di implementare **il paradigma 'divide-et-impera'**, che aiuta a gestire meglio la complessità dei problemi, suddividendoli in problemi più piccoli di dimensioni ridotte.

Esempi?

Programmazione Modulare

Paradigma Divide-et-Impera

La programmazione modulare permette di implementare il **paradigma 'divide-et-impera'**, che aiuta a gestire meglio la complessità dei problemi, suddividendoli in problemi più piccoli di dimensioni ridotte.

Applicato alla programmazione, il paradigma Divide-et-Impera si può applicare sia al **problem solving top-down** (suddivido il problema in sotto-problemi e implemento un sotto-programma per ciascuno di essi).

che quello bottom-up (unisco frammenti di codice più piccoli per implementare programmi complessi)

Programmazione Modulare

- Perché?

Programmazione Modulare

- **Perchè?**

- E' molto difficoltoso creare programmi reali (e complessi) utilizzando **un unico file sorgente**.

Programmazione Modulare

- **Perchè?**

- E' molto difficoltoso creare programmi reali (e complessi) utilizzando **un unico file sorgente**.
 - Difficile tenere traccia **degli errori logici**
 - **Es.** L'output non è quello atteso, quale frammento di codice ha causato il problema?

Programmazione Modulare

- **Perchè?**

- E' molto difficoltoso creare programmi reali (e complessi) utilizzando **un unico file sorgente**.
 - Difficile tenere traccia **degli errori logici**
 - **Es.** L'output non è quello atteso, quale frammento di codice ha causato il problema?
 - Difficile **collaborare** sullo stesso codice
 - **Es.** Come fanno più persone a modificare contemporaneamente porzioni diverse del progetto?

Programmazione Modulare

- **Perchè?**

- E' molto difficoltoso creare programmi reali (e complessi) utilizzando **un unico file sorgente**.
 - Difficile tenere traccia **degli errori logici**
 - **Es.** L'output non è quello atteso, quale frammento di codice ha causato il problema?
 - Difficile **collaborare** sullo stesso codice
 - **Es.** Come fanno più persone a modificare contemporaneamente porzioni diverse del progetto?
 - Difficile **riusare** parti del programma in progetti diversi
 - **Es.** Alcune porzioni di codice sono ricorrenti (calcolo **del Massimo** o della media). Perchè riscrivere il codice varie volte?

Programmazione Modulare

- **Che vantaggi abbiamo?**

- Astrazione
- Riutilizzabilità
- Modularità
- Leggibilità

Programmazione Modulare - Astrazione

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = calcolaBMI(peso, altezza);  
  
    printf("Il tuo BMI è %.2f", bmi);  
}
```

Programmazione Modulare - Astrazione

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = calcolaBMI(peso, altezza); Funzione  
  
    printf("Il tuo BMI è %.2f", bmi);  
}
```

Programmazione Modulare - Astrazione

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = calcolaBMI(peso, altezza); Funzione  
  
    printf("Il tuo BMI è %.2f", bmi);  
}
```

La programmazione modulare aumenta l'**astrazione** del codice sorgente, ci «**nasconde**» i **dettagli implementativi** (come è calcolato il BMI ?) e ci permette di concentrarci su «cosa» il programma deve fare, tralasciando il «come»

Programmazione Modulare - Astrazione

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = calcolaBMI(peso, altezza); Funzione  
  
    printf("Il tuo BMI è %.2f", bmi);  
}
```

Il principio secondo cui è utile «nascondere» dei dettagli implementativi di una funzione prende il nome di **information hiding**, ed è una delle basi della **programmazione ad oggetti**.

Programmazione Modulare - Riutilizzabilità

```
int main() { // Sistema gestionale di una palestra
    . . . // dichiarazione delle variabili omessa

    printf("Inserisci nome e cognome del cliente:");
    scanf("%s %s", nome, cognome);
    printf("Inserisci altezza e peso:");
    scanf("%d %d", &altezza, &peso);
    bmi = calcolaBMI(altezza,peso);

    . . . // dettagli omessi
}
```

Programmazione Modulare - Riusabilità

```
int main() { // Sistema gestionale di una palestra
    . . . // dichiarazione delle variabili omessa

    printf("Inserisci nome e cognome del cliente:");
    scanf("%s %s", nome, cognome);
    printf("Inserisci altezza e peso:");
    scanf("%d %d", &altezza, &peso);
    bmi = calcolaBMI(altezza,peso);

    . . . // dettagli omessi
}
```

L'utilizzo delle funzioni
rende **il codice più
riusabile** perché la stessa
funzione può essere
utilizzata in svariati
programmi diversi

Programmazione Modulare - Modularità

```
int main() { // Sistema gestionale di una palestra
    . . . // dichiarazione delle variabili omessa

    printf("Inserisci nome e cognome del cliente:");
    scanf("%s %s", nome, cognome);
    printf("Inserisci altezza e peso:");
    scanf("%d %d", &altezza, &peso);
    bmi = calcolaBMI(altezza, peso);

    . . . // dettagli omessi
}
```

L'utilizzo delle funzioni rende il **codice più modulare** perché frammenti diversi di codice svolgono un ruolo diverso *(e possono essere testati singolarmente!)*

Programmazione Modulare - Modularità

```
int main() { // Sistema gestionale di una palestra
    . . . // dichiarazione delle variabili omessa

    printf("Inserisci nome e cognome del cliente:");
    scanf("%s %s", nome, cognome);
    printf("Inserisci altezza e peso:");
    scanf("%d %d", &altezza, &peso);
    bmi = calcolaBMI(altezza,peso);

    . . . // dettagli omissi
}
```

Modularità: Una parte del codice acquisisce i dati in input, una parte del codice mostra i risultati, una parte elabora i dati

Programmazione Modulare - Leggibilità

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = (float) peso / ( (altezza/100)*(altezza/100) )  
  
    printf("Il tuo BMI è %.2f",bmi);  
}
```

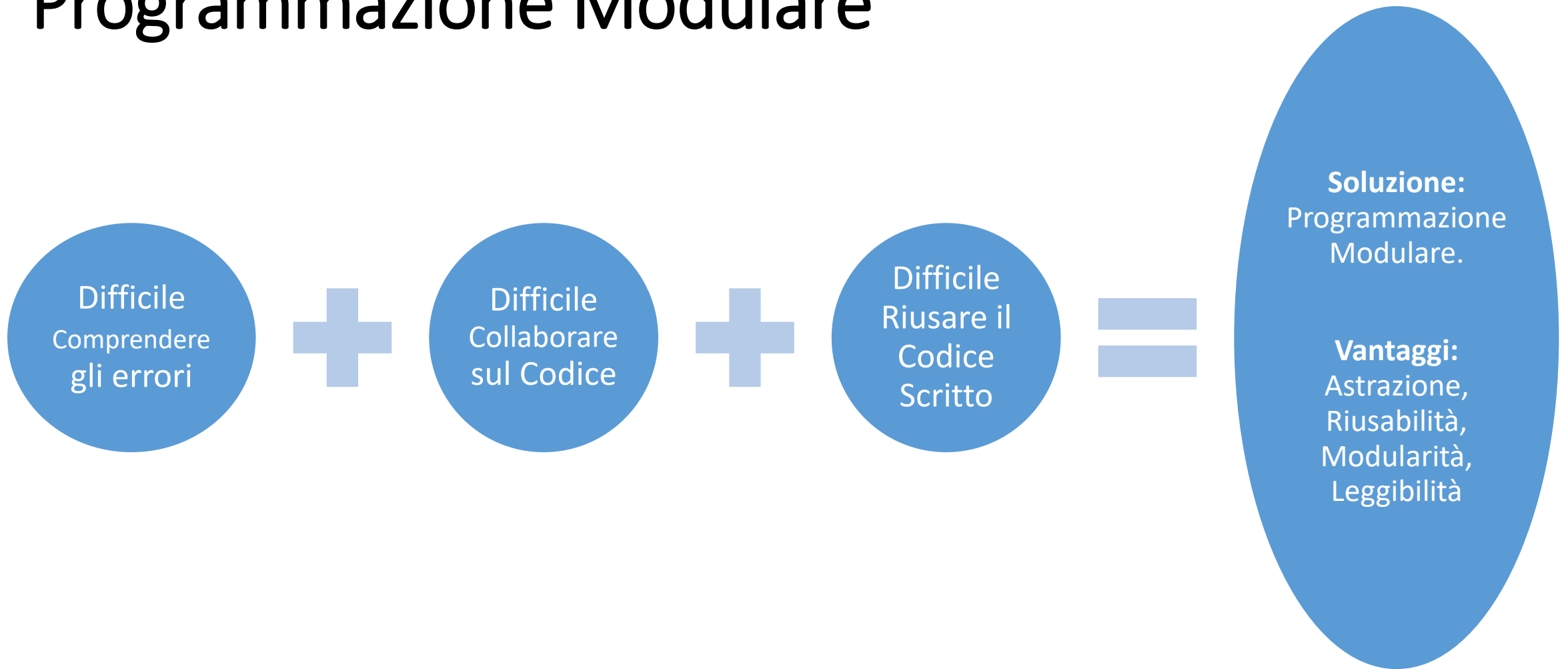
**Questo codice è meno
«leggibile» a chi lo guarda
per la prima volta,
soprattutto se ad esempio
non sa cosa sia il BMI!**

Programmazione Modulare - Leggibilità

```
int main() {  
    int peso = 90;  
    int altezza = 198;  
    float bmi = calcolaBMI(peso, altezza)  
  
    printf("Il tuo BMI è %.2f", bmi);  
}
```

Questo codice
è più leggibile!

Programmazione Modulare



Programmazione Modulare

Principi

- 1) Un programma deve essere suddiviso **in moduli**.
- 2) I moduli devono essere quanto più possibile **indipendenti uno dall'altro**
- 3) Ciascun modulo deve poter essere **testato singolarmente e autonomamente**
- 4) Ogni modulo deve avere **un'interfaccia** (input richiesti/output prodotti) **chiara e ben definite**

Programmazione Modulare

Principi

- 1) Un programma deve essere suddiviso **in moduli**.
 - 2) I moduli devono essere quanto più possibile **indipendenti uno dall'altro**
 - 3) Ciascun modulo deve poter essere **testato singolarmente e autonomamente**
 - 4) Ogni modulo deve avere **un'interfaccia** (input richiesti/output prodotti) **chiara e ben definita**
-
- La Programmazione modulare in Linguaggio C è resa possibile attraverso la **definizione di funzioni e procedure**.

Programmazione Modulare

Principi

- 1) Un programma deve essere suddiviso **in moduli**.
 - 2) I moduli devono essere quanto più possibile **indipendenti uno dall'altro**
 - 3) Ciascun modulo deve poter essere **testato singolarmente e autonomamente**
 - 4) Ogni modulo deve avere **un'interfaccia** (input richiesti/output prodotti) **chiara e ben definita**
-
- La Programmazione modulare in Linguaggio C è resa possibile attraverso la **definizione di funzioni e procedure**.
 - Funzioni e Procedure possono **essere raggruppate in librerie**, che possono essere incluse nei programmi attraverso la direttiva **#include**

Programmazione Modulare

Principi

- 1) Un programma deve essere suddiviso **in moduli**.
- 2) I moduli devono essere quanto più possibile **indipendenti uno dall'altro**
- 3) Ciascun modulo deve poter essere **testato singolarmente e autonomamente**
- 4) Ogni modulo deve avere **un'interfaccia** (input richiesti/output prodotti) **chiara e ben definita**

- **Negli esercizi svolti, in realtà, abbiamo già applicato i principi della Programmazione modulare. Quando?**


Programmazione Modulare

Quando utilizziamo **la direttiva #include** stiamo in realtà già applicando i principi della **programmazione modulare**, perché **stiamo riutilizzando funzioni non previste nel C standard** che sono state scritte da altri programmatori.

Programmazione Modulare

Quando utilizziamo **la direttiva #include** stiamo in realtà già applicando i principi della **programmazione modulare**, perché **stiamo riutilizzando funzioni non previste nel C standard** che sono state scritte da altri programmatori.

```
#include <string.h>

int main() {
    char c = 'a';
    if (isdigit(c)) 
        printf(«%c is a digit», c);
}
```

Progettazione Modulare

- La programmazione modulare richiede un grosso sforzo **di progettazione modulare**, per capire quali parti del programma gestire attraverso sotto-programmi **(e capire ovviamente come implementarli)**

Progettazione Modulare

- La programmazione modulare richiede un grosso sforzo **di progettazione modulare**, per capire quali parti del programma gestire attraverso sotto-programmi **(e capire ovviamente come implementarli)**
- **Attraverso la progettazione modulare applichiamo due principi di programmazione**
 - **Separazione delle competenze**
 - Ogni frammento di codice deve svolgere **un ruolo ben preciso**
 - **Information Hiding**
 - **Si nascondono i dettagli implementativi. Quando implementiamo una funzione, avviene in automatico**

Separazione delle Competenze

- Ogni frammento di codice deve svolgere **un ruolo ben preciso**
- **Esempi**
 - **Una funzione per il calcolo del BMI deve calcolare solo il BMI**
 - Una funzione che acquisisce l'input e calcola anche il BMI **non è una funzione corretta!**
 - Una funzione che calcola il BMI e stampa un messaggio del tipo «L'individuo è sovrappeso» **non è una funzione corretta!**

Separazione delle Competenze

```
void calcolaBMI(void) {  
    int altezza = 0;  
    int peso = 0;  
    float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
  
    BMI = peso / ((altezza/100) * (altezza/100));  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}
```

Separazione delle Competenze

```
void calcolaBMI(void) {  
    int altezza = 0;  
    int peso = 0;  
    float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
  
    BMI = peso / ((altezza/100) * (altezza/100));  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}
```

**Non Corretta, perché effettua
operazioni di lettura input,
elaborazione e stampa output**

Separazione delle Competenze

```
int main (void) {  
    int altezza = 0; int peso = 0; float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
    BMI = calcolaBMI(altezza,peso);  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}  
  
float calcolaBMI(int altezza, int peso) {  
    float BMI = peso / ((altezza/100) * (altezza/100));  
    return BMI  
}
```


Separazione delle Competenze

```
int main (void) {  
    int altezza = 0; int peso = 0; float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
    BMI = calcolaBMI(altezza,peso);  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}
```

**Corretta, perché la funzione
effettua UNA SOLA
OPERAZIONE**

```
float calcolaBMI(int altezza, int peso) {  
    float BMI = peso / ((altezza/100) * (altezza/100));  
    return BMI  
}
```

Separazione delle Competenze

```
int main (void) {  
    int altezza = 0; int peso = 0; float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
    BMI = calcolaBMI(altezza,peso);  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}  
  
float calcolaBMI(int altezza, int peso) {  
    float BMI = peso / ((altezza/100) * (altezza/100));  
    return BMI  
}
```

Importante: nei messaggi di richiesta di input indicare sempre il formato di input (es. in centimetri, in metri) e le modalità di inserimento

Separazione delle Competenze

```
int main (void) {  
    int altezza = 0; int peso = 0; float BMI = 0.0;  
    printf(«Inserisci altezza e peso separati da spazio»);  
    scanf(«%d %d», &altezza, &peso);  
    BMI = calcolaBMI(altezza,peso);  
    printf(«Il tuo BMI è: %.2f, BMI»);  
}  
  
float calcolaBMI(int altezza, int peso) {  
    float BMI = peso / ((altezza/100) * (altezza/100));  
    return BMI  
}
```

Importante: bisogna comunque implementare i controlli sulla correttezza dell'input 😊

Riassumendo...

Programmazione Modulare - Riassunto

Problema: Codice complesso, difficile da riusare, difficile trovare gli errori

Soluzione: Programmazione Modulare

Si divide il problema in sotto-problemi (paradigma divide-et-impera) e si implementa una funzione per ciascuno di essi

Vantaggi: codice più leggibile, più modulare, più riusabile. Maggiore astrazione e generalità del codice.

Importante: fare attenzione al processo di progettazione modulare, cioè **capire quali sono i moduli che devono comporre il programma**, facendo in modo che siano indipendenti e con delle interfacce ben definite

Obiettivo: separazione delle competenze e information hiding



Funzioni e Procedure

In C

Funzioni e Procedure

- Cosa è?
 - I moderni Linguaggi di Programmazione offrono la possibilità di suddividere i programmi **in sotto-programmi, frammenti di codice più piccoli, distinti dal programma principale (main())**
 - Questi frammenti di codice prendono il nome **di funzioni (o di procedure)**
 - Ogni Linguaggio di Programmazione definisce dei meccanismi per **definire** dei sotto-programmi e dei meccanismi per **invocarli**.

Funzioni e Procedure

- Cosa è?
 - I moderni Linguaggi di Programmazione offrono la possibilità di suddividere i programmi **in sotto-programmi, frammenti di codice più piccoli, distinti dal programma principale (main())**
 - Questi frammenti di codice prendono il nome **di funzioni (o di procedure)**
 - Ogni Linguaggio di Programmazione definisce dei meccanismi per **definire** dei sotto-programmi e dei meccanismi per **invocarli**.
- Quali meccanismi di mette a disposizione il linguaggio C ?

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Un insieme di parametri
 - Un valore di ritorno
 - Una implementazione
- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.

- Un nome
- Un insieme di parametri
- Un valore di ritorno
- Una implementazione



- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Meccanismo con cui invochiamo la funzione (o la procedura) in un programma.
 - **Esempio:** `isdigit(c)`
 - Un insieme di parametri
 - Un valore di ritorno
 - Una implementazione
- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Un insieme di parametri
 - Quali sono le variabili di cui ha bisogno la funzione per eseguire il proprio compito
 - **Esempio:** `isdigit(c)` → parametro di tipo 'char'
 - Un valore di ritorno
 - Una implementazione
- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Un insieme di parametri
 - Un valore di ritorno
 - Il valore che ci 'restituisce' la funzione quando termina la sua esecuzione
 - **Esempio:** `isdigit(c)` → restituisce un intero (0/1)
 - Una implementazione
- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Un insieme di parametri
 - Un valore di ritorno
 - Una implementazione
 - Le effettive istruzioni che vengono eseguite. Per il principio dell'information hiding, sono note **solo a chi scrive la funzione**
- I primi tre elementi prendono il nome di **prototipo della funzione**

Funzioni e Procedure

- Ogni funzione (o procedura) è caratterizzata da **quattro elementi**.
 - Un nome
 - Un insieme di parametri
 - Un valore di ritorno
 - Una implementazione
 - Le effettive istruzioni che vengono eseguite. Per il principio dell'information hiding, sono note **solo a chi scrive la funzione**
- I primi tre elementi prendono il nome di **prototipo della funzione**

Attenti a non confondere il prototipo della funzione con la chiamata della funzione!

Funzioni e Procedure

- Prototipo di Funzione
 - `int calcolaBMI(int , int)`
- **Chiamata a funzione**
 - **`calcolaBMI(altezza, peso)`**

Funzioni e Procedure

- Prototipo di Funzione
 - `int calcolaBMI(int , int)`
- **Chiamata a funzione**
 - **`calcolaBMI(altezza, peso)`**
- Il prototipo di funzione serve a illustrare tipo di ritorno, nome e parametri. **La chiamata è un'istanziatura del prototipo, cioè «leghiamo» i parametri a dei nomi di variabile di quel tipo specifico**

Funzioni e Procedure

Sintassi in Linguaggio C

```
<valore_di_ritorno> <nome_della_funzione> (<parametri>) {  
    <implementazione della funzione>  
}
```

Funzioni e Procedure

Sintassi in Linguaggio C

```
<valore_di_ritorno> <nome_della_funzione> (<parametri>) {  
    <implementazione della funzione>  
}
```

Esempio

```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```

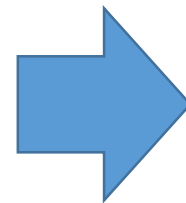
Funzioni e Procedure

Sintassi in Linguaggio C

```
<valore_di_ritorno> <nome_della_funzione> (<parametri>) {  
    <implementazione della funzione>  
}
```

Esempio

```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



Come si «legge» il prototipo di questa funzione?

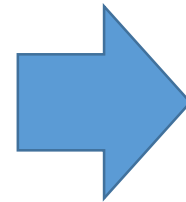
Funzioni e Procedure

Sintassi in Linguaggio C

```
<valore_di_ritorno> <nome_della_funzione> (<parametri>) {  
    <implementazione della funzione>  
}
```

Esempio

```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



La funzione «**promosso**»
prende in input un
parametro di tipo intero e
restituisce un intero

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    printf(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x>18)  
        printf(«Promosso»);  
    else  
        printf(«Non promosso»);  
}
```

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    printf(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x>18)  
        printf(«Promosso»);  
    else  
        printf(«Non promosso»);  
}
```

**Programma in C che non usa
sotto-programmi.**

**E' corretto? Si.
Che problemi ha (secondo i
principi della Programmazione
Modulare)?**

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    printf(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x > 18)  
        printf(«Promosso»);  
    else  
        printf(«Non promosso»);  
}
```

No separazione delle competenze. Un unico metodo effettua due operazioni diverse (Acquisizione dell'input e stampa dell'output)

Funzioni e Procedure – Come usarle

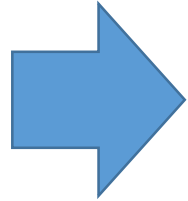
```
int main() {  
    int x = 0;  
    printf(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x>18)  
        printf(«Promosso»);  
    else  
        printf(«Non promosso»);  
}
```

No separazione delle competenze. Un unico metodo effettua due operazioni diverse (Acquisizione dell'input e stampa dell'output)

No information hiding. I dettagli su come venga calcolato l'output non sono «nascosti»

Funzioni e Procedure – Come usarle

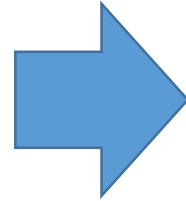
```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x>18)  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}
```



```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}  
  
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}
```

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(x > 18)  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}
```



```
int promosso (int voto) {  
    if(voto > 18) return 1;  
    else return 0;  
}  
  
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}
```


Funzione

**Programma
principale**

Come cambia l'esecuzione dei programmi
quando invochiamo delle funzioni?

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```




Quando il programma legge **promosso(x)** passa il controllo dal programma principale (main) alla funzione.

Nota: per leggibilità, in alcuni esempi la dichiarazione della funzione avviene DOPO il main. **In realtà tutte le funzioni devono precedere il main.**

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```

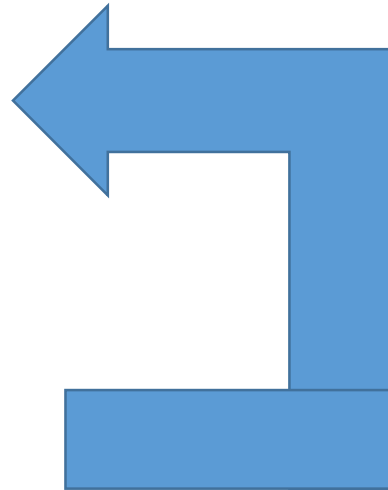


Le istruzioni vengono sempre eseguite sequenzialmente. Quando il programma legge promosso(x) passa il controllo dal programma principale (main) alla funzione.

La funzione effettua i suoi calcoli legando il **parametro formale (voto)** al **parametro attuale (x)**

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



Le istruzioni vengono sempre eseguite sequenzialmente. Quando il programma legge `promosso(x)` passa il controllo dal programma principale (`main`) alla funzione

La funzione effettua i suoi calcoli legando il **parametro formale (voto)** al **parametro attuale (x)**

Quando la funzione ha terminato i suoi calcoli, restituisce un valore (**return**) e il controllo torna alla funzione chiamante

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```

← X → parametro attuale

← voto → parametro formale

Note

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```

← X → parametro attuale

← voto → parametro formale

Note

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Quando la funzione viene chiamata, parametro **formale** e parametro **attuale** si «legano»

Funzioni e Procedure – Come usarle

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}  
  
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```

← X → parametro attuale

← voto → parametro formale

Note

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Quando la funzione viene chiamata, parametro **formale** e parametro **attuale** si «legano»

I tipi del parametro formale e di quello attuale devono essere compatibile. Altrimenti si genera un **errore di compilazione**

Funzioni e Procedure – Come usarle

```
int main() {  
    int x_vet[] = {0};  
    puts(«Inserisci voto:»);  
    scanf(«%d»,x_vet[0]);  
  
    if(promosso(x_vet))  
        puts(«Promosso»);  
    else  
        puts(«Non promosso»);  
}
```



Tipi non
compatibili

```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



Errore di
compilazione

Note

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Quando la funzione viene chiamata, parametro **formale** e parametro **attuale** si «legano»

I tipi del parametro formale e di quello attuale devono essere compatibile. Altrimenti si genera un **errore di compilazione**

Funzioni e Procedure – Prototipi di Funzione

```
int promosso (int voto); // prototipo di funzione
```

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x)) puts(«Promosso»);  
    else puts(«Non promosso»);  
}
```

```
// implementazione della funzione
```

```
int promosso (int voto) {  
    if(voto > 18) return 1;  
    else return 0;  
}
```

E' utile dichiarare tutte le funzioni
PRIMA del main.

E' OBBLIGATORIO COMMENTARLE,
spiegando che scopo svolgono

Queste dichiarazioni si chiamano
prototipi di funzione

Funzioni e Procedure – Prototipi di Funzione

```
int promosso (int voto); // prototipo di funzione
```

```
int main() {  
    int x = 0;  
    puts(«Inserisci voto:»);  
    scanf(«%d», &x);  
  
    if(promosso(x)) puts(«Promosso»);  
    else puts(«Non promosso»);  
}
```

```
// implementazione della funzione
```

```
int promosso (int voto) {  
    if(voto > 18) return 1;  
    else return 0;  
}
```

Importante: per
migliorare la leggibilità
del codice utilizzare
sempre nomi
significativi per funzioni
e **soprattutto parametri**

E' utile dichiarare tutte le funzioni
PRIMA del main.

E' OBBLIGATORIO COMMENTARLE,
spiegando che scopo svolgono

Queste dichiarazioni si chiamano
prototipi di funzione

Aumentano la leggibilità del codice

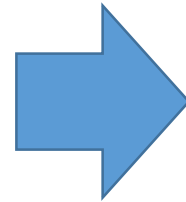
Servono al compilatore a controllare
che la funzione venga **chiamata nel**
modo corretto.

Importante: non esiste un'unica
intestazione delle funzioni
(e quindi un'unica implementazione)

Funzioni e Procedure

Esempio

```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



La funzione «**promosso**»
prende in input un
parametro di tipo intero e
restituisce un intero

E' corretta? Si!

E' l'unica implementazione possibile? NO

Funzioni e Procedure

Implementazione Alternativa

```
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

Implementazione
alternativa, **ancora più
generale**

Funzioni e Procedure

Implementazione Alternativa

```
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

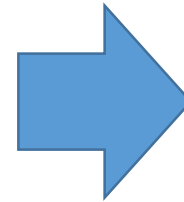
Implementazione
alternativa, ancora più
generale

Questa implementazione è ancora più generale! Mi permette di utilizzare la funzione in diversi ambiti (es. esami di laurea, di maturità, scuola guida, etc.) → **riusabilità del codice**

Funzioni e Procedure

Esempio

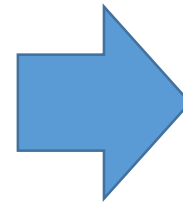
```
int promosso (int voto) {  
    if(voto>18) return 1;  
    else return 0;  
}
```



La funzione «**promosso**»
prende in input un
parametro di tipo intero e
restituisce un intero

Implementazione Alternativa

```
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```



Implementazione
alternativa, **ancora più
generale**

La scelta dei parametri di input di una funzione e la tipologia di valori che deve restituire è un elemento fondamentale di progettazione modulare.

Funzioni e Procedure – Come usarle

```
int main() {  
    int x, y = 0;  
    puts(«Inserisci voto ESAME:»);  
    scanf(«%d», &x);  
    puts(«Inserisci voto MATURITA':»);  
    scanf(«%d», &y);  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}
```

Aggiungendo un solo parametro alla funzione, la rendo molto più generale e applicabile a diversi scenari.

E' fondamentale riflettere attentamente su quale sia lo scopo di una funzione e quali siano gli scenari in cui applicarla

Funzioni e Procedure

- Finora funzioni e procedure sono stati usati come sinonimi.
- **C'è differenza?**

Funzioni e Procedure

- Finora funzioni e procedure sono stati usati come sinonimi.
- **C'è differenza?**
- **Una procedura è una funzione che non restituisce alcun valore**
 - In C si utilizza la parola chiave **void** per indicare che la funzione non restituisce nulla

Funzioni e Procedure

- Finora funzioni e procedure sono stati usati come sinonimi.
- **C'è differenza?**
- **Una procedura è una funzione che non restituisce alcun valore**
 - In C si utilizza la parola chiave **void** per indicare che la funzione non restituisce nulla

```
int main() {  
    x = 20;  
    stampaNumero(x); //procedura  
}  
  
void stampaNumero (int numero) {  
    printf(«%d», numero);  
}
```


Funzioni e Procedure

- Finora funzioni e procedure sono stati usati come sinonimi.
- **C'è differenza?**
- **Una procedura è una funzione che non restituisce alcun valore**
 - In C si utilizza la parola chiave **void** per indicare che la funzione non restituisce nulla

```
int main() {  
    x = 20;  
    stampaNumero(x); //procedura  
}  
  
void stampaNumero (int numero) {  
    printf(«%d», numero);  
}
```

Non c'è return.
Non c'è valore di ritorno (c'è void).

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

Di nuovo...

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Quando la funzione viene chiamata, parametro **formale** e parametro **attuale** si «legano»

I tipi del parametro formale e di quello attuale devono essere compatibile. Altrimenti si genera un **errore di compilazione**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

Di nuovo...

Si dice «**parametro formale**» quello definito nell'intestazione (o nel prototipo) della funzione

Si dice **parametro attuale** quello definito nella chiamata alla funzione

Quando la funzione viene chiamata, parametro **formale** e parametro **attuale** si «legano»

Il meccanismo con cui parametro formale e attuale si legano si dice **passaggio dei parametri**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di
passaggio dei parametri

Passaggio Per Valore

Passaggio Per Riferimento

Di default, il **passaggio dei
parametri avviene per valore.**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {
    int x = 22;  int y = 50;

    if(promosso(x, 18))
        puts(«Promosso»);
    else puts(«Non promosso»);

    if(promosso(y, 60))
        puts(«Promosso»);
    else puts(«Non promosso»);
}

int promosso (int voto, int soglia) {
    if(voto>soglia) return 1;
    else return 0;
}
```

In C esistono due modalità di
passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore
effettua una copia del valore del
parametro **attuale** nel parametro
formale

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di
passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore
effettua una copia del valore del
parametro **attuale** nel parametro
formale

x

y

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22; int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x = 22

y = 50

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x	= 22
y	= 50

il controllo passa alla funzione!

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x = 22

y = 50

voto

soglia

**Si alloca
memoria per le
due variabili
locali**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x = 22

y = 50

voto = 22

soglia = 18

Il valore viene **copiato** → **passaggio dei parametri**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x = 22

y = 50

Usciti dalla
funzione **la
memoria viene
liberata!**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di passaggio dei parametri

Passaggio Per Valore

Il passaggio dei parametri per valore **effettua una copia** del valore del parametro **attuale** nel parametro **formale**

x	= 22
y	= 50
voto	= 50
soglia	= 60

Ala seconda chiamata della funzione, viene **effettuata una nuova copia**.

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 22;  int y = 50;  
  
    if(promosso(x, 18))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
  
    if(promosso(y, 60))  
        puts(«Promosso»);  
    else puts(«Non promosso»);  
}  
  
int promosso (int voto, int soglia) {  
    if(voto>soglia) return 1;  
    else return 0;  
}
```

In C esistono due modalità di
passaggio dei parametri

Passaggio Per Valore

Quali sono i problemi del passaggio per
valore?

Eventuali modifiche ai parametri
vengono perse nel quando l'esecuzione
della funzione termina.

Vediamo un esempio.

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

Fuori: 5

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

Fuori: 5
Dentro: 25

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

Fuori: 5
Dentro: 25
Fuori: 5

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

Fuori: 5
Dentro: 25
Fuori: 5

Il valore di X viene modificato dalla funzione (infatti dentro la funzione il valore è raddoppiato).

Terminata l'esecuzione, le variabili locali alla funzione vengono distrutte, **e i risultati si perdono.**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}  
  
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Cosa stampa questo programma?

```
Fuori: 5  
Dentro: 25  
Fuori: 5
```

Per evitare di «perdere» i risultati elaborati dalle funzioni **si utilizza la seconda tipologia di passaggio dei parametri, il passaggio per riferimento.**

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
`main()`

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Ogni funzione (compreso il **main()**) ha la sua area dati.

Significa **che il compilatore destina uno spazio di memoria** sufficiente ad allocare gestire le variabili del programma.

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()

X

5

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```


Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()



```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Nel momento in cui la sequenza delle istruzioni giunge ad una chiamata a funzione, **si crea una nuova area dati.**

In questo caso è dedicata alla funzione **quadrato()**

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()



```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Area Dati
quadrato(int)

```
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()

X	5
---	---

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Area Dati
quadrato(int)

Y	5
---	---

```
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()



```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Area Dati
quadrato(int)



```
void quadrato (int y) {  
    y = y*y  
    printf(«Dentro: %d», y);  
}
```

A white arrow points from the 'y' in the assignment statement 'y = y*y' to the 'y' in the printf statement 'printf(«Dentro: %d», y);'.

Passaggio per valore: immaginate di «copiare» il valore della variabile in un'altra locazione di memoria. La funzione lavora sulla «nuova» locazione di memoria, la vecchia non viene mai modificata.

Area Dati
main()

X 5

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(x);  
    printf(«Fuori: %d», x);  
}
```

Tutte le modifiche vengono operate **nell'area dati della funzione (non del main!)**

Terminata l'esecuzione della funzione, **tutto il contenuto dell'area dati viene perso.**

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x[3] = {1,2,3}; // array di int  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}  
// restituisce il quadrato  
void quadrato (int x[], int n) {  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```

Il passaggio dei parametri avviene di **default per valore**.

Fanno eccezione gli array. Gli array vengono passati per riferimento. Ciò significa **che le modifiche operate a un array si riflettono anche fuori dalla funzione**.

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x[3] = {1,2,3}; // array di int  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}  
// restituisce il quadrato  
void quadrato (int x[], int n) {  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```

Il passaggio dei parametri avviene di **default per valore**.

Fanno eccezione gli array. Gli array vengono passati per riferimento. Ciò significa **che le modifiche operate a un array si riflettono anche fuori dalla funzione**.

Cosa stampa questo programma?

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x[3] = {1,2,3}; // array di int  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}  
// restituisce il quadrato  
void quadrato (int x[], int n) {  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```

Il passaggio dei parametri avviene di **default per valore**.

Fanno eccezione gli array. Gli array vengono passati per riferimento. Ciò significa **che le modifiche operate a un array si riflettono anche fuori dalla funzione**.

Cosa stampa questo programma?

1 2 3

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x[3] = {1,2,3}; // array di int  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}  
// restituisce il quadrato  
void quadrato (int x[], int n) {  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```

Il passaggio dei parametri avviene di **default per valore**.

Fanno eccezione gli array. Gli array vengono passati per riferimento. Ciò significa **che le modifiche operate a un array si riflettono anche fuori dalla funzione**.

Cosa stampa questo programma?

1	2	3
1	4	9

Funzioni e Procedure – Passaggio dei Parametri

```
int main() {  
    int x[3] = {1,2,3}; // array di int  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}  
  
void quadrato (int x[], int n)  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```

Quando si passa un array a una funzione, bisogna ricordarsi di passare anche la sua dimensione

Il passaggio dei parametri avviene di **default per valore**.

Fanno eccezione gli array. Gli array vengono passati per riferimento. Ciò significa **che le modifiche operate a un array si riflettono anche fuori dalla funzione**.

Cosa stampa questo programma?

1	2	3
1	4	9

Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Area Dati
main()




```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(&x);  
    printf(«Fuori: %d», x);  
}
```




Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Area Dati
main()

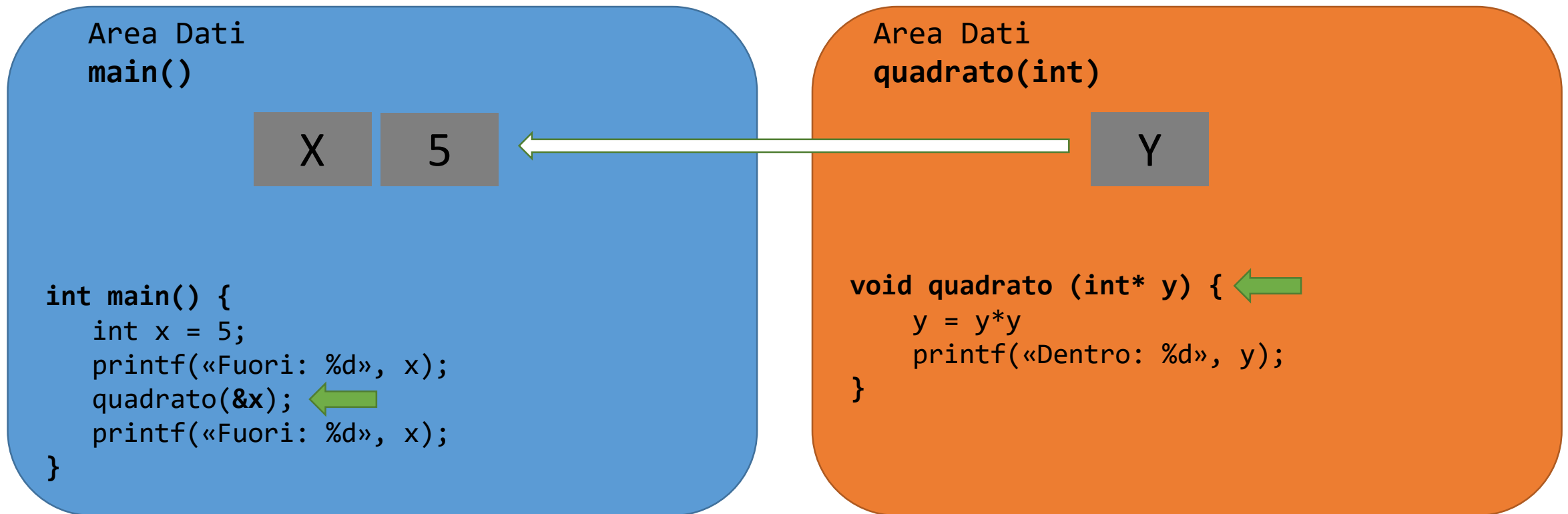


```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(&x);   
    printf(«Fuori: %d», x);  
}
```

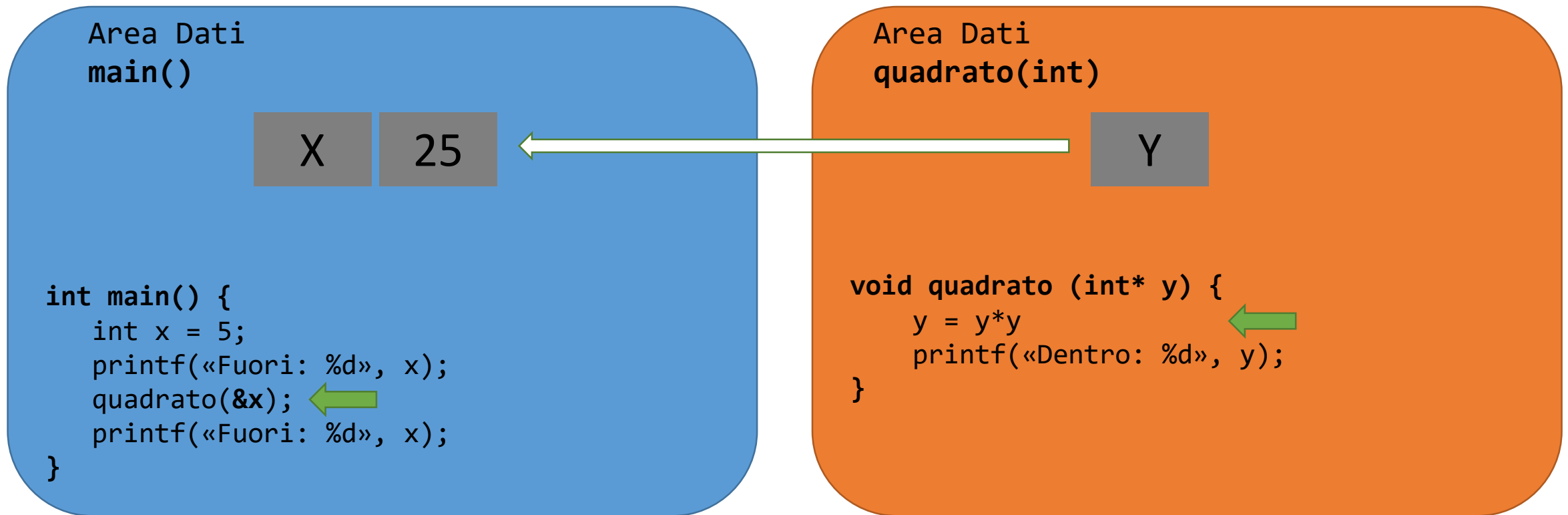
Area Dati
quadrato(int)

```
void quadrato (int* y) {   
    y = y*y;  
    printf(«Dentro: %d», y);  
}
```

Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**



Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**



Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Area Dati
main()

X 25

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(&x);  
    printf(«Fuori: %d», x);  
}
```

Tutte le modifiche vengono ereditate **dal main**

Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Area Dati
main()

X 25

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(&x);  
    printf(«Fuori: %d», x);  
}
```

Tutte le modifiche vengono ereditate **dal main**

Cosa stampa questo programma?

Fuori: 5
Dentro: 25
Fuori: 25 ←

Passaggio per riferimento: sia il parametro formale che il parametro attuale puntano alla stessa locazione di memoria. Le modifiche quindi non vengono perse, **anzi, vengono ereditate dal programma chiamante.**

Area Dati
main()

X

25

```
int main() {  
    int x = 5;  
    printf(«Fuori: %d», x);  
    quadrato(&x);  
    printf(«Fuori: %d», x);  
}
```

Tutte le modifiche vengono ereditate **dal main**

Chiaramente per effettuare un passaggio dei parametri per riferimento bisogna «passare» l'indirizzo di una variabile (&x), non il suo valore!

Programmazione Modulare

Concetti Chiave

Programmazione Modulare - Recap

- 1) Per aumentare modularità, separazione delle competenze, riusabilità, leggibilità e information hiding è **utile dividere il programma in più sotto-programmi**
- 2) Ciascun sotto-programma deve avere uno scopo ben definito e **svolgere una sola funzione**

Programmazione Modulare - Recap

- 3) Bisogna pensare attentamente (progettazione modulare) a
- **Quali porzioni del programma originale dividere in sotto-programmi**
 - **Cosa deve fare** ciascun sotto-programma
 - Quali **dati di input** ha bisogno (parametri) , quali **dati restituisce** (return)
 - **IMPORTANTE:** non esiste un'unica intestazione di una funzione. E' una importante scelta di progetto. Quanto più ampio è il numero di parametro, **tanto più generale diventa la funzione.**
- 4) Quando una funzione viene chiamata, il compilatore effettua dei controlli sui **parametri formali e i parametri attuali.**
- Viene verificato il numero, il tipo e l'ordine dei parametri. **Se non sono corretti, viene restituito un errore di compilazione.** Se sono corretti, i parametri si «legano» e si passa il controllo alla funzione

Programmazione Modulare – Recap (Cont.)

5) Al termine dell'esecuzione della funzione **si ripassa il controllo al programma chiamante** e **tutte le variabili locali alla funzione vengono distrutte.**

6) Nel passaggio per valore, parametri formali e attuali vengono «legati» attraverso una copia del valore (che poi viene perso). **Nel passaggio per riferimento** invece parametro formale attuale lavorano sulla stessa locazione di memoria (quindi il risultato non viene perso)

- Tutte le variabili vengono passate **per default per valore**
- Gli array vengono passati per default **per riferimento.**
- I puntatori (valuteremo in seguito pro e contro) permettono il **passaggio per riferimento.**

Esercizio 6.1

- **Migliorare l'esercizio relativo al calcolo del BMI (Esercitazione 0) utilizzando la programmazione modulare**
 - 1) Il programma deve memorizzare in un vettore il BMI di cinque individui.
 - 2) Il BMI deve essere generato random a partire da valori casuali di altezza e peso (altezza tra 160 e 200, peso tra 60 e 120). Stampa i valori generati.
 - 3) Calcolare BMI medio dei cinque individui. Calcolare BMI massimo.
 - 4) Stampare messaggio relativo al BMI di ogni individuo («è sottopeso», «è sovrappeso»). Stampa BMI massimo e BMI medio.

Note: definire le funzioni che si vogliono implementare nel main. Dichiarare i prototipi prima del main.

Esercizio 6.1

- **Quali funzioni prevediamo?**
 - Una funzione per calcolare il BMI
 - Una funzione per calcolare la media
 - Una funzione per calcolare il massimo
- **Eventualmente**
 - Una funzione per l'acquisizione dell'input
 - Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**
 - Una funzione per calcolare il BMI
 - Una funzione per calcolare la media
 - Una funzione per calcolare il massimo
 - Una funzione per l'acquisizione dell'input
 - Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**
 - Una funzione per calcolare il BMI
 - `float calculateBMI(int weight, int height)`
 - Una funzione per calcolare la media
 - Una funzione per calcolare il massimo
 - Una funzione per l'acquisizione dell'input
 - Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**
 - Una funzione per calcolare il BMI
 - `float calculateBMI(int weight, int height)`
 - Una funzione per calcolare la media
 - `float calculateAverage(float values[], int n)`
 - Una funzione per calcolare il massimo
 - Una funzione per l'acquisizione dell'input
 - Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**

- Una funzione per calcolare il BMI
 - `float calculateBMI(int weight, int height)`
- Una funzione per calcolare la media
 - `float calculateAverage(float values[], int n)`
- Una funzione per calcolare il massimo
 - `float calculateMaximum(float values[], int n)`
- Una funzione per l'acquisizione dell'input
- Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**
 - Una funzione per calcolare il BMI
 - `float calculateBMI(int weight, int height)`
 - Una funzione per calcolare la media
 - `float calculateAverage(float values[], int n)`
 - Una funzione per calcolare il massimo
 - `float calculateMaximum(float values[], int n)`
 - Una funzione per l'acquisizione dell'input
 - `float acquireInput(float values[], int n)`
 - Una funzione per la stampa dell'output

Esercizio 6.1

- **Quali sono i prototipi di funzione?**
 - Una funzione per calcolare il BMI
 - `float calculateBMI(int weight, int height)`
 - Una funzione per calcolare la media
 - `float calculateAverage(float values[], int n)`
 - Una funzione per calcolare il massimo
 - `float calculateMaximum(float values[], int n)`
 - Una funzione per l'acquisizione dell'input
 - `float acquireInput(float values[], int n)`
 - Una funzione per la stampa dell'output
 - `float printOutput(float values[], int n, float average, float maximum)`

Esercizio 6.1 - Soluzione

```
int main(void) {  
    float BMIs[DIMENSION] = {0.0}; // inizializzo a zero l'array dei BMI  
  
    acquireInput(BMIs, DIMENSION); // acquisisco l'input  
    float average = calculateAverage(BMIs, DIMENSION); // calcolo la media  
    float maximum = calculateMaximum(BMIs, DIMENSION); // calcolo il massimo  
    printOutput(BMIs, DIMENSION, average, maximum); // stampo l'output  
  
    return 0;  
}
```

**Nella programmazione modulare
IDEALE, il main dovrebbe contenere
SOLO chiamate a funzione.**

Esercizio 6.1 - Soluzione

```
int main(void) {  
    float BMIs[DIMENSION] = {0.0}; // inizializzo a zero l'array dei BMI  
  
    ➡ acquireInput(BMIs, DIMENSION); // acquisisco l'input  
    float average = calculateAverage(BMIs, DIMENSION); // calcolo la media  
    float maximum = calculateMaximum(BMIs, DIMENSION); // calcolo il massimo  
    ➡ printOutput(BMIs, DIMENSION, average, maximum); // stampo l'output  
  
    return 0;  
}
```

Le fasi di acquisizione e stampa possono anche essere inserite nel `main()`, senza incapsularle in una procedura. **Il livello di modularità scende, ma è comunque una corretta implementazione**

Esercizio 6.1 – Soluzione (cont.)

```
void acquireInput( float array[], int size );  
  
float calculateBMI( int height, int weight );  
  
float calculateAverage( float array[], int size );  
  
float calculateMaximum( float array[], int size );  
  
void printOutput( float array[], int size, float average, float  
maximum );
```

Si dichiarano i prototipi di funzione (si può scegliere se commentarli nella dichiarazione dei prototipi o prima dell'implementazione)

Esercizio 6.1 – Soluzione (cont.)

```
// procedura di acquisizione dell'input: prende in input un array di valori e  
// restituisce void
```

```
void acquireInput( float array[], int size ) {  
    int seed = time(NULL);  
    srand(seed);  
    int height = 0;  
    int weight = 0;  
    float BMI = 0.0; // dichiaro le variabili locali per altezza, peso e BMI  
  
    for(unsigned int i=0; i<DIMENSION; i++) {  
        height = rand() % 41 + 160;  
        weight = rand() % 61 + 40;  
        BMI = calculateBMI(height, weight); // calcolo BMI  
        array[i] = BMI; // memorizzo il valore calcolato nell'array  
    }  
}
```

Esercizio 6.1 – Soluzione (cont.)

```
// funzione per il calcolo del BMI: prende in input una coppia di  
valori (altezza e peso) e restituisce il BMI corrispondente
```

```
float calculateBMI( int height, int weight ) {  
    float BMI = (float) weight /  
                (((float)height/100) * ((float)height/100));  
  
    return BMI;  
}
```

Esercizio 6.1 – Soluzione (cont.)

```
// funzione di calcolo della media: prende in input un array di valori e
// restituisce il valore medio

float calculateAverage( float array[], int size ) {
    float average = 0.0; // dichiarare sempre la variabile locale da
                          // restituire. In questo caso la variabile
                          // funge anche da variabile di appoggio per la
                          // somma.

    for(unsigned int i=0; i<DIMENSION; i++) {
        average = average + array[i];
    }

    average /= size; // ottengo la media
    return average;
}
```

Esercizio 6.1 – Soluzione (cont.)

```
// funzione di calcolo del massimo: prende in input un array di valori e  
restituisce il valore massimo
```

```
float calculateMaximum( float array[], int size ) {  
    float maximum = 0.0; // dichiarare sempre la variabile locale da  
                          // restituire.  
  
    for(unsigned int i=0; i<DIMENSION; i++) {  
        if ( array[i] > maximum)  
            maximum = array[i]; // se il valore letto è maggiore  
                                // del massimo attuale, aggiorno  
                                // il massimo  
    }  
  
    return maximum;  
}
```

Esercizio 6.1 – Soluzione (cont.)

```
// procedura per la stampa dell'output: prende in input tutti i valori da
// stampare e li mostra in output

void printOutput( float array[], int size, float average, float maximum ) {
    for (unsigned int i=0; i < DIMENSION; i++) {
        printf("Individuo %d, BMI: %.2f ", i+1, array[i]);

        if(array[i] > BMI_THRESHOLD)          // stampa di un messaggio a
            printf(" è sovrappeso\n"); // seconda del BMI
        else printf(" non è sovrappeso\n");
    }

    printf("Il valore medio del BMI è %.2f\n", average);
    printf("Il valore massimo del BMI è %.2f\n", maximum);
}
```

