

# Passaggio dei parametri

---

## ■ Per valore

- Il valore viene ***copiato*** dall' *environment* esterno all' *environment* della funzione o procedura
- ***Cambiamenti*** dei parametri così passati ***non si riflettono*** sull' *environment* esterno

## ■ Per riferimento (o indirizzo)

- Non esiste a livello di linguaggio in C ma può essere implementato direttamente dal programmatore tramite ***puntatori***
- Viene ***copiato (per valore) l'indirizzo della variabile*** da passare
- ***Cambiamenti*** dei parametri così passati ***si riflettono*** sull' *environment* esterno → **attenzione a cambiare i valori e non gli indirizzi**

# Passaggio dei parametri

---

- Il passaggio avviene **formalmente sempre per valore**
- Sta al programmatore scegliere se vuole passare (per valore) alla funzione/procedura il valore o l'indirizzo del contenitore

```
void setSeven(int *pi)
{
    *pi = 7;
}
```

*Pretende un  
puntatore ad intero*

```
int main()
{
    int i = 0;
    setSeven(&i);
    printf("%d", i);
}
```

*Dereferenzia ed usa il  
contenitore puntato dal  
puntatore*

*Passa l'indirizzo della  
variabile → passa un  
puntatore a intero*

# Passaggio dei parametri


---

- Altri linguaggi (C++, C#, Java, ...) forniscono il passaggio di parametri per riferimento

Per esempio in C#

```
static void SetSeven(ref int i)
{
    i = 7;
}
```

```
public static void Main(...)
{
    int i = 0;
    SetSeven(ref i);
    Console.Write(i);
}
```



Se si omette *ref*  
si ha errore di  
**compilazione**

# Passaggio dei parametri

---

- Normalmente usare il passaggio per valore
- Usare il passaggio per riferimento in casi particolari
  1. La funzione/procedura deve **restituire più di un valore**  

```
int getTwoValues(int *value1, int *value2);
```
  2. **Non è conveniente** passare “il dato” per valore:
    - Caso di strutture dati “voluminose” → si vedranno più avanti...
  3. **Non è possibile** passare “il dato” per valore:
    - Array (quindi anche stringhe)
    - Strutture dati allocate dinamicamente → si vedranno più avanti...
- Nei casi 2 e 3, attenzione agli effetti collaterali!

# Passaggio dei parametri

---

Che cosa succede se:

```
void setSeven(int *pi)
{
    *pi = 7;
}

int main()
{
    int i = 0;
    setSeven(i);
    printf("%d", i);
}
```

- Anziché passare l'indirizzo della variabile `i`, viene passato il valore di `i`
- La procedura usa il valore come puntatore → il compilatore segnala solo un warning...
- Errore a runtime (Bad Pointer exception) → tentativo di accesso alla locazione di memoria `0x00000000` (`i` vale 0)

# Equazioni di secondo grado

---

Data l'equazione  $ax^2 + bx + c = 0$ , se ne calcolino le radici

- Comportamento in caso d'errore (delta negativo)?
- Procedura o funzione?
- Quali parametri?

# Equazioni di secondo grado

---

- Comportamento in caso d' errore?
  - La chiamata “fallisce” restituendo un valore di controllo che indica l' errore
- Procedura o funzione?
  - Quanti valori da restituire?
    - 2 radici
    - 1 controllo d' errore (delta negativo)
  - Possibile scelta: funzione
    - Parametro di ritorno → controllo d' errore (vero, falso)
    - Radici restituite tramite parametri passati per indirizzo
- Quali parametri?
  - Quelli decisi sopra (2 radici) più coefficienti equazione (a, b, c) passati per valore

# Nota sul controllo d' errore

---

- Le funzioni che possono fallire devono poter restituire un valore di controllo...
- ...che indica se la funzione è stata eseguita con successo e, in caso negativo, indica il tipo d' errore
- Come controllo può essere usato:
  - valore restituito dalla funzione
  - valore inserito in una variabile globale
  - ...l' invocazione di una funzione di gestione dell' errore, possibilmente definita dall' utente (*ma come si potrebbe fare?*)



# Nota sul controllo d' errore

---

- **printf ()** può fallire?
  - Non “gentilmente”; gli errori di protezione sono sempre in agguato ☹
- **scanf ()** può fallire?
  - Sì → es: si richiede un intero, l'utente inserisce una stringa alfanumerica
  - Il valore di ritorno della **scanf** indica **quanti parametri sono stati letti correttamente**
    - **N == 0** → insuccesso
    - **N == n. parametri da leggere** → successo
    - **N != n. parametri da leggere** → ...

# Equazioni di secondo grado

---

## ■ Interfaccia della funzione

```
#define BOOLEAN int
#define TRUE 1
#define FALSE 0
BOOLEAN solve(int a, int b, int c,
               float *r1, float *r2);
```

- **a, b, c** sono i coefficienti dell'equazione
- **r1, r2** contengono gli indirizzi delle variabili dove scrivere le radici
- Il valore di ritorno contiene il codice d'errore

# Equazioni di secondo grado

---

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

#define BOOLEAN int
#define TRUE 1
#define FALSE 0

BOOLEAN solve(int a, int b, int c, float *r1, float *r2);

int main()
{
    int a, b, c;
    float x1, x2;
    scanf("%d %d %d\n", &a, &b, &c);    //Controllo d'errore?
    if ( solve(a, b, c, &x1, &x2) )
        printf("x1 = %f; x2 = %f\n", x1, x2);
    else
        printf("Errore: delta negativo!");
}
```

# Equazioni di secondo grado

---

```
BOOLEAN solve(int a, int b, int C, float *r1, float
    *r2)
{
    float delta;
    delta = b * b - 4 * a * c;
    if (delta < 0)
        return FALSE;
    else
    {
        *r1 = (-b + sqrt(delta)) / (2 * a);
        *r2 = (-b - sqrt(delta)) / (2 * a);
        return TRUE;
    }
}
```