C++: funzioni

Docente: Ing. Edoardo Fusella
Dipartimento di Ingegneria Elettrica e Tecnologie dell’Informazione
Via Claudio 21, 4° piano – laboratorio SECLAB
Università degli Studi di Napoli Federico II
e-mail: edoardo.fusella@unina.it

Funzioni: cosa sono

- Le funzioni rappresentano dei contenitori per dei frammenti di codice
- Esse possono essere invocate ogni qualvolta si desideri eseguire uno specifico frammento
- I frammenti di codice che costituiscono una funzione vengono spesso definiti “sotto-programmi” (sub-routines)
- Una chiamata a funzione provoca:
  - l’esecuzione del codice contenuto nella funzione;
  - l’esecuzione della istruzione del programma chiamante immediatamente successiva alla chiamata a funzione, una volta che l’esecuzione del codice della funzione è terminata

Esempio di programma che utilizza funzioni

```
//Programma che, utilizzando funzioni, realizza
//l'ordinamento di un vettore inserito dall'utente
int main() {
  TVettore V;
  int nelem;

  CaricaVettore(…);  //Chiamata a funz.

  cout << "Il vettore non ordinato e': " << endl;
  StampaVettore(…); //Chiamata a funz.

  OrdinaVettore(…); //Chiamata a funz.

  cout << "Il vettore ordinato e': " << endl;
  StampaVettore(…); //Chiamata a funz.
}
```

Vantaggi nell'uso delle funzioni

Con riferimento al programma d'esempio notiamo che l'uso delle funzioni apporta alcuni vantaggi:

- **maggiore sinteticità**
  - una funzione può essere scritta una sola volta ed usata tante volte
    - Si consideri l'esempio della funzione StampaVettore(…)
  - programmi più brevi sono più facilmente manutenibili!
- **maggiore leggibilità**
  - da un veloce sguardo del codice, è subito chiaro "cosa fa" il programma
- **possibilità di scomporre ogni problema in sottoproblemi più semplici**
  - si può realizzare il programma in più fasi successive, non dovendo preoccuparsi di "tutto e subito"
  - si può delegare la stesura di parti di programma ad altri individui
- **possibilità di concentrarsi più sulla logica del problema** che sull'implementazione della sua soluzione
  - pur sapendo "cosa fa" una funzione, non è detto che si sappia "come lo fa"
    - Si consideri l'esempio della funzione OrdinaVettore()

Esempio: la funzione somma

senza le funzioni

```
int main() {
  int a, b;
  int s;

  cout << "Inserisci a: ";
  cin >> a;
  cout << "Inserisci b: ";
  cin >> b;

  s = a + b;

  cout << "La somma vale: " << s;
  cout << endl;
}
```

con le funzioni

```
int Somma(int x, int y); //prototipo

int main() {
  int a, b;
  int s;

  cout << "Inserisci a: ";
  cin >> a;
  cout << "Inserisci b: ";
  cin >> b;

  s = Somma(a, b);

  cout << "La somma vale: " << s;
  cout << endl;
}

int Somma(int x, int y) {
  return x + y;
}
```

Il prototipo di una funzione

Il prototipo di una funzione definisce:

- il nome della funzione
- il tipo di parametri che tratta, siano essi di ingresso o di uscita o di ingresso/uscita
- Il tipo del valore eventualmente restituito

<tipo risult.> <NomeFunz.>(<tipo> <nome>, <tipo> <nome>, …);

Per esempio:

- bool DataValida(int giorno, int mese, int anno);
- bool NumeroPari(int n);
- float Dividi(float a, float b);

Valori restituiti

- Dal prototipo di una funzione è possibile dedurre il tipo del valore che assumerà la funzione in seguito alla sua chiamata

- Il valore assunto dalla funzione può essere utilizzato, per es., all'interno di espressioni più complesse:

  d = Somma(a,b) * Dividi(a,c); // d=(a+b)*a/c

- L'argomento stesso di una funzione potrebbe essere il risultato restituito da un'altra funzione

  e = Somma(a, Somma(b, c)); // e=a+b+c

Procedure

Una funzione può anche non restituire alcun valore.

- In questo caso il suo prototipo sarà del tipo:

void Func(…);

- Le funzioni che non restituiscono alcunché, vengono anche dette procedure, ma in C++ la differenza tra funzioni e procedure resta molto blanda
- La chiamata ad una procedura è del tipo

Func(…);

- … e non ha più senso qualcosa del tipo

~~**a = Func(…);**~~

- Esempi di procedure potrebbero essere:

– StampaVettore(…)

– CaricaVettore(…)

– AzzeraIntero(…)

– …

La keyword “return”

Dall’interno della funzione, per restituire il giusto valore al chiamante, si usa la keyword ***return***

- Essa, come effetto collaterale, provoca anche l’immediata terminazione della funzione
- Esempio:

```
//funzione che indica se c è compreso nell’intervallo
//a..b estremi inclusi (implementazione poco felice)
bool  Compreso(int  a, int b, int c) {
if (c < a) return false;
if (c > b) return false;
return true;
}
```

- La stessa funzione può essere più felicemente realizzata nel modo seguente:

```
bool  Compreso(int  a, int b, int c) {
return ( (c >= a) && (c <= b) );
}
```

Parametri formali e parametri effettivi (1/3)

- Nel prototipo della funzione Compreso():

  bool Compreso(int a, int b, int c)

  a, b e c sono detti **parametri formali**
  - Sono i parametri che la funzione utilizza al suo interno
  - È come se fosse stata la funzione ad averli dichiarati

- Al momento della definizione della funzione non è necessario conoscere quali saranno i veri parametri su cui essa opererà

Parametri formali e parametri effettivi (2/3)

- Nel programma

```
int main() {
    int x, y, z;
    ....
    if (Compreso(x, y, z)) {
        cout << "z e' compreso tra x ed y.\n";
    }
    else {
        cout << "z non e' compreso tra x ed y.\n";
    }
    return 0;
}
```

- all'atto della chiamata alla funzione, x, y e z sono detti **parametri effettivi**, e non sono tenuti ad avere lo stesso nome dei parametri formali

Parametri formali e parametri effettivi (3/3)

- All’atto della chiamata:
  - il compilatore controlla se il tipo dei parametri effettivi corrisponde col tipo dei parametri formali, segnalando eventualmente un errore;
  - sostituisce ciascun parametro formale con il corrispondente parametro effettivo;
  - esegue la funzione.
- **La corrispondenza non avviene per nome, ma <u>per posizione</u>**
  - In altre parole, al primo parametro formale viene sostituito il primo parametro effettivo, al secondo parametro formale il secondo parametro effettivo, e così via

La funzione main

- Anche il main() è una funzione
- La sua unica caratteristica peculiare è che, per convenzione, **è la prima funzione invocata in un programma**
- È il sistema operativo ad invocare la funzione main() ed a recuperare, al suo termine, il valore restituito attraverso il return
- Si noti anche che, così come è possibile dichiarare delle variabili locali al main() utili al suo funzionamento, è ugualmente possibile dichiarare delle variabili all'interno di tutte le altre funzioni

Esempio 1

- Scrivere il prototipo di una funzione che accetti in ingresso un intero e, interpretandolo come un anno, restituisca un valore che indichi se l'anno specificato è bisestile.
- Si implementi poi la funzione.
- Si realizzi infine un programma chiamante (main) che verifichi il corretto comportamento della funzione

Esempio 2

- Scrivere il prototipo di una funzione che accetti in ingresso tre interi e, interpretandoli come giorno, mese ed anno, restituisca un valore che indichi se la data specificata è una data valida.
- Si implementi poi la funzione.
- Si realizzi infine un programma chiamante (main) che verifichi il corretto comportamento della funzione

Il passaggio dei parametri per valore (1/4)

- Si consideri il seguente programma con il relativo output:

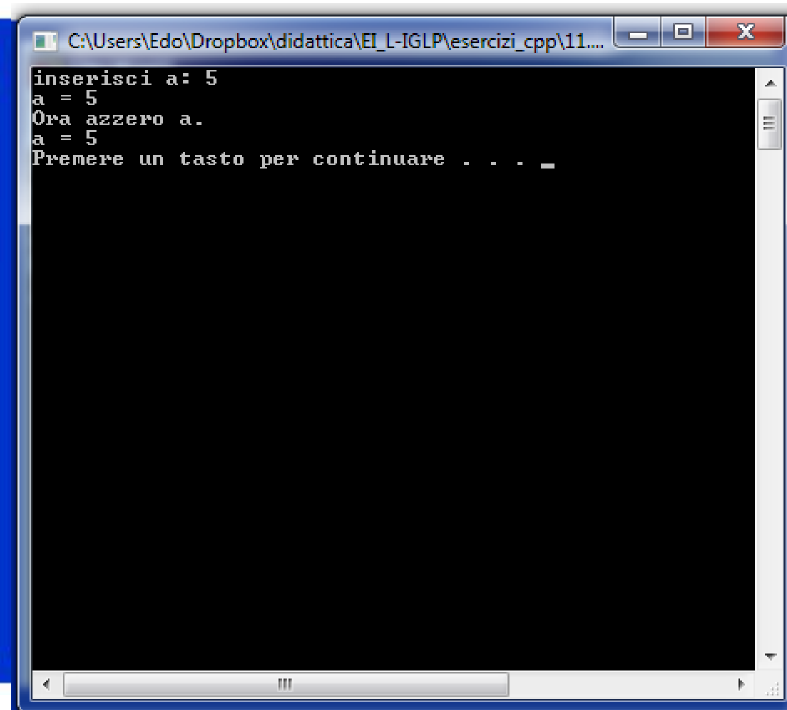
```
void Azzera(int numero);

int main() {
  int a;
  cout << "Inserisci a: ";
  cin >> a;
  cout << "a = " << a << endl;

  cout << "Ora azzero a.";
  Azzera(a);

  cout << "a = " << a << endl;
}

void Azzera(int numero) {
  numero = 0;
}
```



- Questo comportamento apparentemente strano si spiega attraverso la modalità di passaggio dei parametri impiegata in questo caso particolare

Il passaggio dei parametri per valore (2/4)

- Il passaggio dei parametri in C++, in assenza di esplicite direttive, avviene secondo la modalità detta per valore
- Questo significa che, all’atto della chiamata di una funzione, il compilatore realizza **una copia dei parametri effettivi** e la associa ai parametri formali
- la funzione lavora dunque su delle copie dei parametri effettivi localizzate in aree di memoria completamente diverse, e non sui parametri effettivi veri e propri
- Le copie vengono distrutte al termine della funzione: del loro valore, eventualmente alterato all’interno della funzione, non resta traccia

Il passaggio dei parametri per valore (3/4)

- Vantaggi:
  - dal punto di vista del programma chiamante:
    - il chiamante di una funzione può essere certo che i parametri ad essa passati non potranno essere alterati in seguito alla chiamata
  - dal punto di vista della funzione:
    - la funzione, se lo crede opportuno, può modificare i parametri a suo piacimento con la certezza che le modifiche non saranno visibili all'esterno di essa
- Svantaggi:
  - l'occupazione di memoria risulta doppia rispetto al necessario
  - il tempo per effettuare la copia dei parametri, specialmente nel caso in cui questi siano appartenenti a tipi strutturati di grosse dimensioni, può degradare le prestazioni di un programma

Il passaggio dei parametri per valore (4/4)

- Al di là dei vantaggi e svantaggi elencati, ci sono dei casi in cui il comportamento che si ottiene col passaggio per valore è del tutto indesiderato
- Esempio: il caso della funzione Azzera()…

```
void Azzera(int numero) {
    numero = 0;
}
```

  - In questo caso, si desidera che il parametro effettivo **venga modificato** in seguito alla chiamata alla funzione, ma il meccanismo di protezione dovuto al passaggio dei parametri per valore impedisce che questo possa avvenire

Il passaggio dei parametri per riferimento(1/3)

- Per risolvere il problema, è necessario esplicitamente richiedere al compilatore che all'interno della funzione si possa lavorare non su delle copie, ma sui parametri effettivi veri e propri
- Questo è possibile attraverso il passaggio dei parametri per riferimento
- In questo caso, alla funzione viene passata non una copia del parametro attuale, ma il riferimento ad esso, cioè l'indirizzo di memoria
- Per richiedere questo tipo di passaggio, bisogna aggiungere il carattere **'&'** tra il tipo ed il nome del parametro in questione

Il passaggio dei parametri per riferimento(2/3)

- La versione corretta della funzione Azzera risulta dunque:

```
void Azzera(int&  numero) {
     numero = 0;
}
```

- In questo caso, il compilatore permette alla funzione di lavorare direttamente sull'area di memoria in cui è contenuto il parametro effettivo posto in corrispondenza col parametro formale "numero", senza quindi che di esso ne venga fatta una copia indipendente

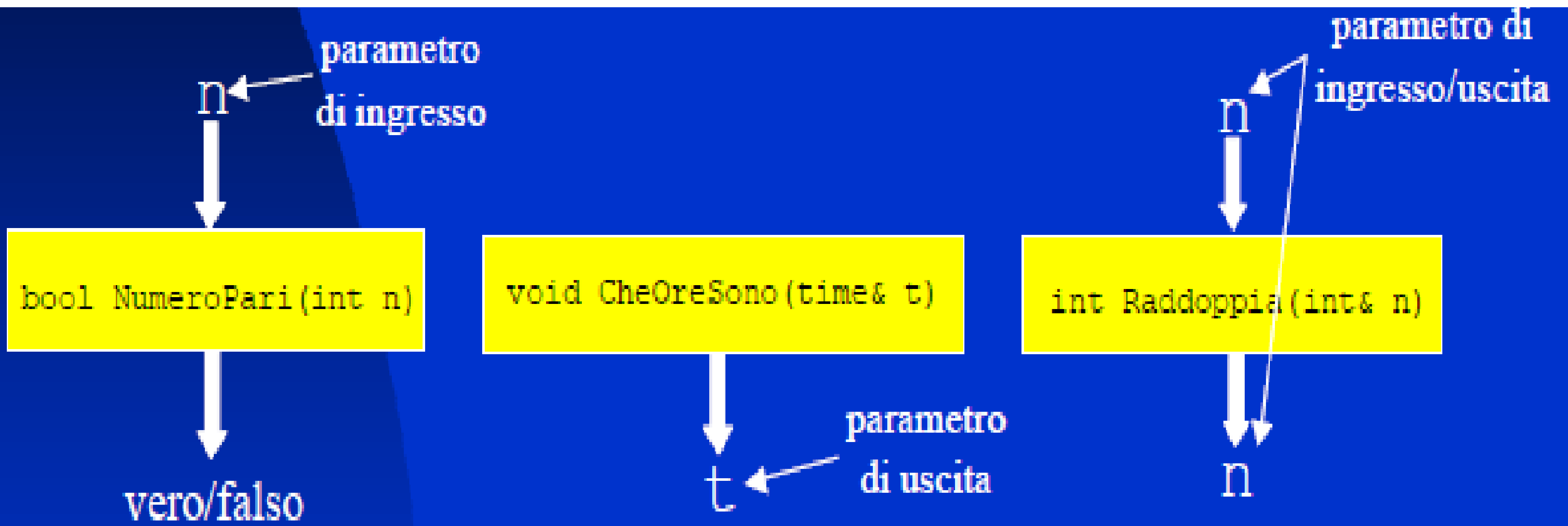
Il passaggio dei parametri per riferimento(3/3)

- Vantaggi:
  - il passaggio è più efficiente dal momento che, a prescindere dalle dimensioni del dato, quello che deve essere passato è sempre e solo un indirizzo di memoria
- Svantaggi:
  - si perde il meccanismo di protezione garantito dal passaggio dei parametri per valore

Valore o riferimento?

- Abbiamo visto che entrambi i tipi di passaggi sono indispensabili per un corretto comportamento delle funzioni in tutti i casi che si possono verificare
- I parametri di una funzione convogliano informazioni tra il chiamante e la funzione chiamata, in entrambe le direzioni
- È possibile individuare il tipo di passaggio che deve essere di volta in volta utilizzato attraverso l'analisi della "direzione" che le informazioni hanno rispetto alla funzione chiamata
- …In altre parole, bisogna capire se le informazioni trasportate dalle variabili di passaggio sono
  - di ingresso alle funzioni chiamate,
  - di ingresso-uscita
  - o di uscita

Parametri di ingresso, di uscita e di ingresso-uscita (1/4)



Parametri di ingresso, di uscita e di ingresso-uscita (2/4)

Definiamo i parametri di una funzione:

- **parametri di ingresso**, se ai fini della corretta esecuzione della funzione è sufficiente, per la funzione stessa, esclusivamente leggere il loro valore;
  - esempio: NumeroPari(n)
- **parametri di uscita**, se essi rappresentano esclusivamente un supporto per convogliare informazioni verso l'esterno della funzione;
  - esempio: CheOreSono(t)
- **parametri di ingresso-uscita**, se il loro valore all'ingresso della funzione è significativo ai fini della elaborazione che essa realizza ma vengono anche alterati per convogliare informazioni verso il chiamante;
  - esempio: Raddoppia(n)

Parametri di ingresso, di uscita e di ingresso-uscita (3/4)

Le precedenti osservazioni ci indicano dunque che:

- i **parametri di ingresso** sono preferibilmente scambiati **per valore**.
- i **parametri di uscita** non possono essere scambiati per valore, altrimenti le modifiche apportate ad essi non sarebbero visibili all’esterno della funzione. Devono dunque essere scambiati **per riferimento**.
- i **parametri di ingresso-uscita**, analogamente a quelli di uscita, non possono essere scambiati per valore, ma devono essere scambiati **per riferimento**.

Parametri di ingresso, di uscita e di ingresso-uscita (4/4)

- Il C++, pertanto, nell'ottica della modalità di passaggio dei parametri, non fa grande differenza tra parametri di uscita e di ingresso-uscita.

- Si tenga presente che una generica funzione può avere più parametri di ingresso, uscita e ingresso-uscita contemporaneamente.

- Esempio:

```
void ModuloFase(float PReale, float PImmag, float& Modulo, float& Fase);
```

Il passaggio di parametri const (1/3)

- E se è necessario passare un dato di grosse dimensioni conservando l’efficienza e proteggendolo comunque da modifiche indesiderate?
- Una soluzione è rappresentata dalla clausola **const**

Il passaggio di parametri const (2/3)

- Durante lo scambio dei parametri, se si fa anticipare al tipo del parametro la keyword **const**, si impedisce del tutto alla funzione di modificare il parametro all'interno di essa.
- Esempi:

int NumeroPari(const int& n);

int QuantiAnniHa(const TPersona& p);

- Nel caso di passaggio per riferimento, la clausola const risolve il problema di modifiche indesiderate ai parametri, consentendo contemporaneamente di sfruttare l'efficienza intrinseca di questa modalità

Il passaggio di parametri const (3/3)

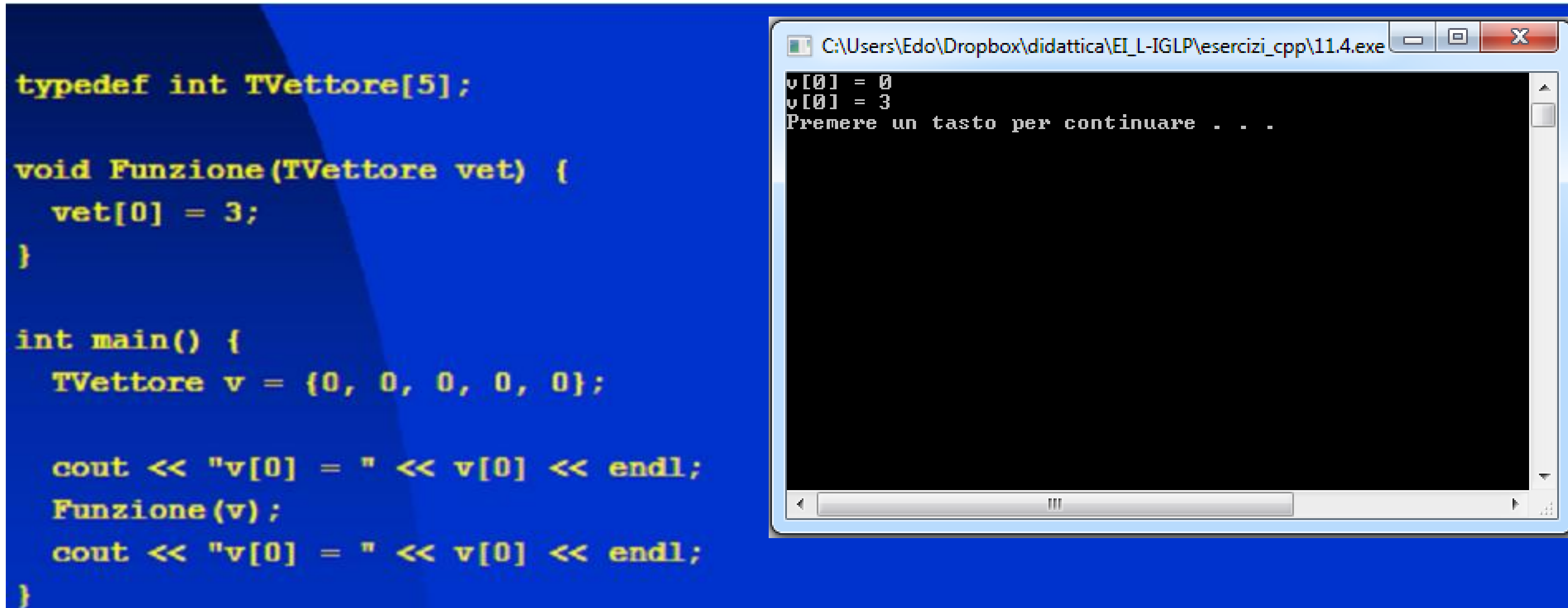
- Il prototipo di una funzione che calcola la radice quadrata di un numero potrebbe essere:

  long double sqrt(long double x);
- All’interno della funzione, se il parametro x viene alterato, le sue modifiche non saranno propagate verso l’esterno
- Un prototipo alternativo potrebbe essere:

  long double sqrt(const long double& x);
- In questo caso l’efficienza aumenta senza compromettere il meccanismo di protezione

Il passaggio dei vettori (1/3)

- Si consideri il seguente programma con il relativo output:



- I vettori, passati come parametri alle funzioni, assumono un comportamento apparentemente strano
- Stando al passaggio dei parametri per valore, l’output riportato di fianco sembra inspiegabile

Il passaggio dei vettori (2/3)

- Questo strano comportamento discende dalla **particolare proprietà dei vettori** secondo la quale il nome di un vettore rappresenta un puntatore alla locazione in cui si trova il primo elemento dell’ array
- Il parametro che quindi viene scambiato per valore, e pertanto viene copiato all’atto del passaggio, non è il vettore ma il suo puntatore
- Ecco perché le modifiche interne alla funzione si propagano anche all’esterno: in entrambi i casi si lavora sempre sulla stessa area di memoria

Il passaggio dei vettori (3/3)

- Per evitare che un parametro di tipo vettore sia alterato da una funzione, bisogna utilizzare la clausola const
- Ciò avviene tipicamente nel caso di funzioni che accettino vettori di ingresso. Esempio:

  void StampaVettore(const TVettore v, int nelem);

- La funzione StampaVettore() in questo modo non può (neanche se volesse) alterare il vettore v
  - Qualsiasi tentativo di farlo produrrebbe un errore di compilazione
- Si noti che l’unico modo per conoscere all’interno di una funzione il numero di elementi significativi di un vettore passato è quello di indicarlo esplicitamente come parametro di passaggio aggiuntivo
  - Il parametro *nelem* nell’esempio rappresenta la dimensione del vettore v

Esempio 3

- Dato un vettore di numeri interi ordinati in ordine crescente
  1. Visualizzare il contenuto del vettore
  2. Trovare in che posizione si trova un dato valore
  3. Cancellare un elemento dal vettore
  4. Inserire (ordinatamente) un valore nel vettore

- Si implementi ogni punto tramite funzioni separate.
- Si realizzi infine un programma chiamante (main) che verifichi il corretto comportamento della funzione