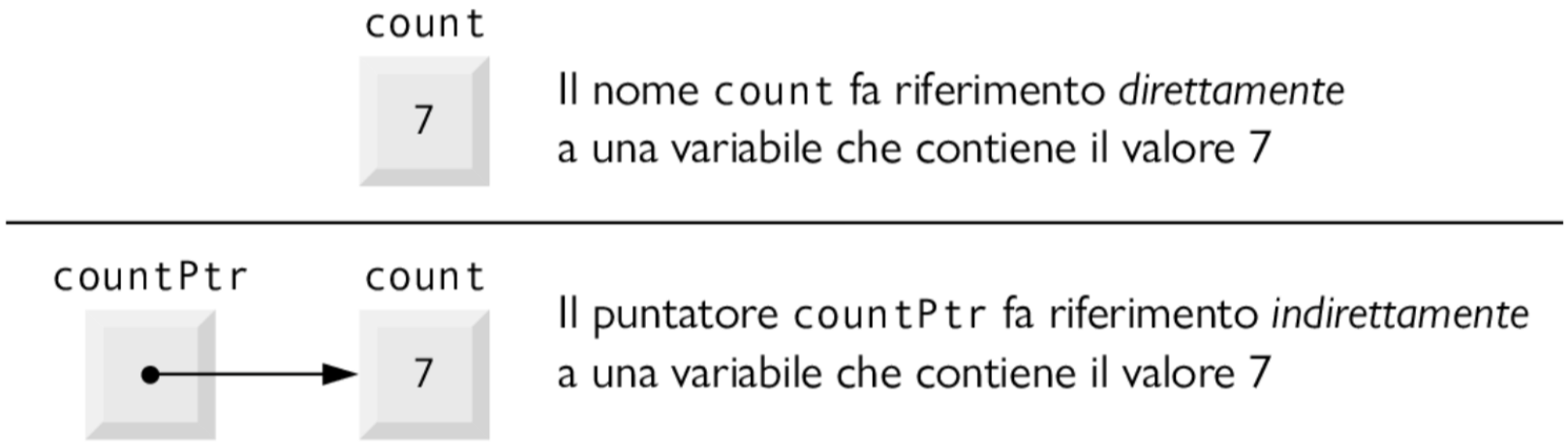
Puntatori

Cosa sono i puntatori

- Sono variabili che contengono **indirizzi di memoria**.
- Una variabile normale contiene direttamente un **valore specifico**
- Un puntatore, invece, contiene l'indirizzo di un'altra variabile che a sua volta contiene un valore specifico.
- Un puntatore "punta" a un'altra variabile

Cosa sono i puntatori



Fonte: Deitel & Deitel

- Il nome di una variabile fa riferimento **diretto** a un valore
- Un puntatore fa riferimento al valore **in modo indiretto**, come mostrato nel diagramma
- Riferirsi a un valore tramite un puntatore si chiama **indirezione**

Definire un puntatore

- Come tutte le variabili, i puntatori devono essere definiti **prima di essere utilizzati**
- Esempio di definizione di un puntatore:

```
int *countPtr;
```

- Si legge da destra a sinistra:
  - `countPtr` è un puntatore a un `int` oppure punta a un oggetto di tipo `int`
- Il simbolo `*` indica che la variabile è un **puntatore**

Variabili puntatore

- Aggiungere `Ptr` alla fine del nome della variabile per indicare che si tratta di un puntatore (es. `countPtr`)
- Altre convenzioni comuni:
  - Iniziare il nome della variabile con `p` (es. `pCount`)
  - Utilizzare un prefisso `p_` (es. `p_count`)
- Definire le variabili in istruzioni separate:

```
int *countPtr, count;
```

- Il simbolo `*` si applica solo a `countPtr`:
  - `countPtr` è un puntatore a un intero (`int`).
  - `count` è semplicemente un intero.
- Per evitare ambiguità, definire le variabili in istruzioni separate:

```
int *countPtr;
int count;
```

Inizializzazione puntatori

- I puntatori devono essere inizializzati al momento della definizione o tramite assegnazione
- Un puntatore può essere inizializzato con:
  - `NULL`
  - `0`
  - Un indirizzo di memoria.
- Un puntatore `NULL` non punta a niente
- `NULL` è una costante simbolica con valore 0, definita in `<stddef.h>` (e in altre intestazioni come `<stdio.h>`)

Inizializzazione puntatori

- Inizializzare un puntatore a `0` è equivalente a inizializzarlo a `NULL`
  - `NULL` evidenzia che state inizializzando un puntatore, non una variabile numerica
  - Quando si assegna `0`, questo viene convertito al tipo di puntatore appropriato
  - `0` è l'unico valore intero che può essere assegnato direttamente a una variabile puntatore.
- Inizializzare i puntatori per prevenire risultati inaspettati

L’operatore &

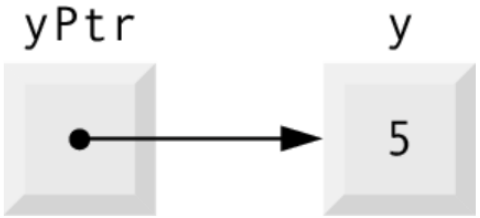
- L'operatore unario & restituisce l'**indirizzo** del suo operando
- Data la seguente definizione:

```
int y = 5;
```

- L'istruzione:

```
int *yPtr = &y;
```

- Inizializza il puntatore `yPtr` con l'indirizzo della variabile `y`
- Si dice che `yPtr` **punta a** `y`
- L’operando di `&` deve essere una variabile; l’operatore di indirizzo non può essere applicato a valori letterali (come 27 o 41.5) o espressioni



Fonte: Deitel & Deitel

Rappresentazione del puntatore in memoria

Fonte: Deitel & Deitel

- Il diagramma mostra la rappresentazione del puntatore precedente in memoria supponendo che
  - la variabile intera `y` sia memorizzata alla locazione `600000`
  - La variabile puntatore `yPtr` sia memorizzata alla locazione `500000`

L’operatore di indirezione *

- L'operatore unario di indirezione (o dereferenziazione) `*` si applica a un puntatore per ottenere l’oggetto **puntato**
- L'istruzione:

```
printf("%d", *yPtr);
```

- Stampa 5, che è il valore della variabile `y` a cui `yPtr` punta.
- Usare * in questo modo equivale a **dereferenziare un puntatore**
- Dereferenziare un puntatore che non è stato inizializzato correttamente o a cui non è stato assegnato un indirizzo valido può causare:
  - **Errori irreversibili** in fase di esecuzione
  - Modifiche accidentali a dati importanti, causando risultati **scorretti**
  - Potenziali **violazioni di sicurezza**

Come funzionano gli operatori & e *

```c
// fig07_01.c
// Uso degli operatori & e *.
#include <stdio.h>

int main(void) {
   int a = 7;
   int  *aPtr = &a; // imposta aPtr all'indirizzo di a

   printf("Address of a is %p\nValue of aPtr is %p\n\n", &a, aPtr);
   printf("Value of a is %d\nValue of *aPtr is %d\n\n", a, *aPtr);
   printf("Showing that * and & are complements of each other\n");
    printf("&*aPtr = %p\n*&aPtr = %p\n", &*aPtr, *&aPtr);
}
```

```
Address of a is 0x7fffe69386cc
Value of aPtr is 0x7fffe69386cc
Value of a is 7
Value of *aPtr is 7
Showing that * and & are complements of each other
&*aPtr = 0x7fffe69386cc
*&aPtr = 0x7fffe69386cc
```

Fonte: Deitel & Deitel

Come funzionano gli operatori `&` e `*`

- Specifica di conversione `%p` di `printf`
  - Stampa l'indirizzo di memoria come un intero esadecimale sulla maggior parte delle piattaforme
- Il programma mostra che l'indirizzo di `a` e il valore di `aPtr` sono identici:
  - Conferma che l'indirizzo di `a` è stato assegnato alla variabile puntatore `aPtr`
- Gli operatori `&` e `*` sono complementari:
  - Quando applicati consecutivamente a `aPtr` (in qualsiasi ordine), il risultato stampato è lo stesso
- Gli indirizzi mostrati nell'output cambieranno nei vari sistemi che utilizzano architetture del processore, compilatori e persino impostazioni del compilatore differenti

Quiz

Passare argomenti alle funzioni

- **Passaggio per valore**: per default, tutti gli argomenti (eccetto gli array) sono passati per valore
- **Passaggio per riferimento**: gli array sono passati per riferimento
  - Permette alle funzioni di modificare le variabili nella funzione chiamante
  - Evita il sovraccarico di memoria evitando di copiare oggetti di grandi dimensioni
  - Consente a una funzione di "restituire" più valori modificando le variabili della funzione chiamante
- Limite del passaggio per valore:
  - Un'istruzione `return` può restituire al massimo un valore da una funzione chiamata alla sua chiamante

Passaggio per riferimento con * e &

- Puntatori e operatore di indirezione (*):
  - Consentono di realizzare il passaggio per riferimento
  - Permettono di modificare direttamente le variabili nella funzione chiamante
- Quando si chiama una funzione e si desidera modificare gli argomenti nella funzione chiamante:
    - Si usa l'operatore & per passare l’indirizzo di ogni variabile

Passaggio per riferimento con * e &

- Gli array non richiedono l'uso dell'operatore &:
    - Il nome di un array è equivalente a &arrayNome[0], ovvero l'indirizzo iniziale dell'array in memoria
- Una funzione che riceve l'indirizzo di una variabile:
    - Può usare l'operatore di indirezione (*) per modificare il valore all'indirizzo specificato
    - Questo effettua il passaggio per riferimento

Calcolo del cubo: passaggio per valore

```c
// fig07_02.c
// Calcolo del cubo di una variabile usando il passaggio per valore.
#include <stdio.h>

int cubeByValue(int n); // prototipo

int main(void) {
   int number = 5; // inizializza number

   printf("The original value of number is %d", number);
   number = cubeByValue(number); // passa number per valore a cubeByValue
   printf("\nThe new value of number is %d\n", number);
}

// calcola e restituisci il cubo di un argomento intero
int cubeByValue(int n) {
   return n * n * n; // restituisci il cubo di n
}
```

```
The original value of number is 5
The new value of number is 125
```

Fonte: Deitel & Deitel

Calcolo del cubo: passaggio per riferimento

```c
// fig07_03.c
// Calcolo del cubo di una variabile usando il passaggio per riferimento.

#include <stdio.h>

void  cubeByReference( int  *nPtr); // prototipo di funzione

int main(void) {
   int number = 5; // inizializza number

   printf("The original value of number is %d", number);
   cubeByReference(&number); // passa l'indirizzo di number a cubeByReference
   printf("\nThe new value of number is %d\n", number);
 }

 // eleva al cubo *nPtr; di fatto modifica number in main
 void cubeByReference( int  *nPtr ) {
    *nPtr = *nPtr * *nPtr * *nPtr; // calcola il cubo di *nPtr
 }
```

```
The original value of number is 5
The new value of number is 125
```

Fonte: Deitel & Deitel

Calcolo del cubo: passaggio per riferimento

- L'indirizzo della variabile `number` viene passato alla funzione `cubeByReference` (passaggio per riferimento)
- Il parametro della funzione è un puntatore a un `int` chiamato `nPtr`
- La funzione utilizza `*nPtr` per dereferenziare il puntatore e elevare al cubo il valore a cui punta
- Il risultato è assegnato a `*nPtr`, modificando così il valore della variabile `number` nella funzione `main`
- Consigliato utilizzare il passaggio per valore a meno che la funzione chiamante non richieda esplicitamente che la funzione chiamata modifichi il valore della variabile argomento
- Questo previene:
  - modifiche accidentali degli argomenti passati dalla funzione chiamante
  - violazioni del principio del privilegio minimo

Usare un parametro puntatore

- Una funzione che riceve un indirizzo come argomento deve usare un **parametro puntatore**

```
void cubeByReference(int *nPtr)
```

- Specifica che `cubeByReference` riceve l'indirizzo di una variabile intera come argomento
- Memorizza l'indirizzo localmente nel parametro `nPtr` e non restituisce alcun valore
- Non è necessario includere i nomi dei puntatori nei prototipi di funzioni
- I nomi dei puntatori vengono ignorati dal compilatore
- È comunque buona pratica includerli per fini di documentazione

Passare un array unidimensionale

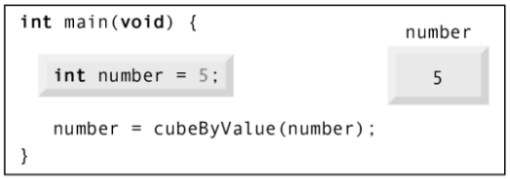
- Il prototipo e l'intestazione di una funzione possono usare la notazione dei puntatori per array unidimensionali

```
void cubeByReference(int *nPtr)
```

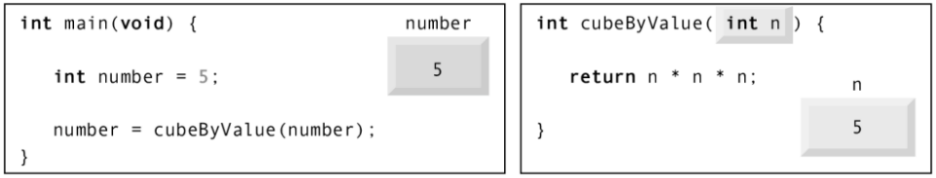
- La funzione `cubeByReference` usa `int *nPtr` per ricevere un array unidimensionale
- Il compilatore non distingue tra una funzione che riceve un **puntatore** e una che riceve un **array unidimensionale**
  - Il compilatore tratta un parametro di tipo `int b[]` come `int *b`
- La funzione deve sapere se sta ricevendo un array o una variabile singola passata per riferimento
- Quando il compilatore incontra un parametro di funzione per un array unidimensionale converte `int b[]` in `int *b`

Analisi di un tipico passaggio per valore

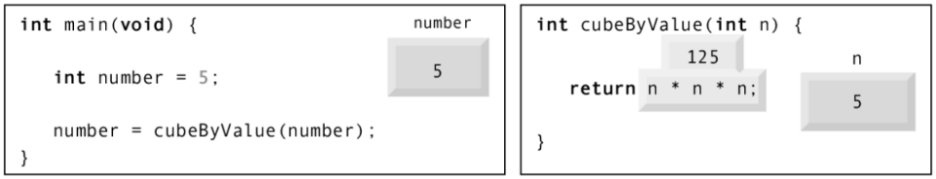
Passo 1: Prima che `main` chiami `cubeByValue`:



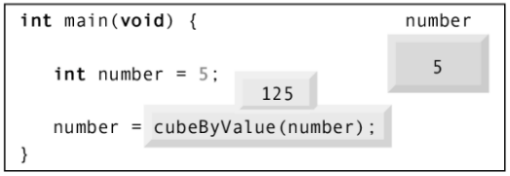
Passo 2: Dopo che `cubeByValue` ha ricevuto la chiamata:



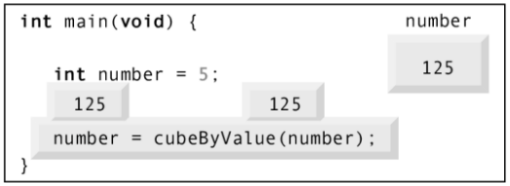
Passo 3: Dopo che `cubeByValue` ha elevato al cubo il parametro `n` e prima che `cubeByValue` torni alla funzione `main`:



Passo 4: Dopo che `cubeByValue` è tornata alla funzione `main` e prima che si assegni il risultato a `number`:



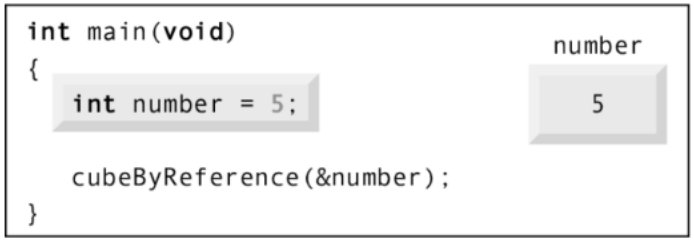
Passo 5: Dopo che `main` ha completato l'assegnazione a `number`:



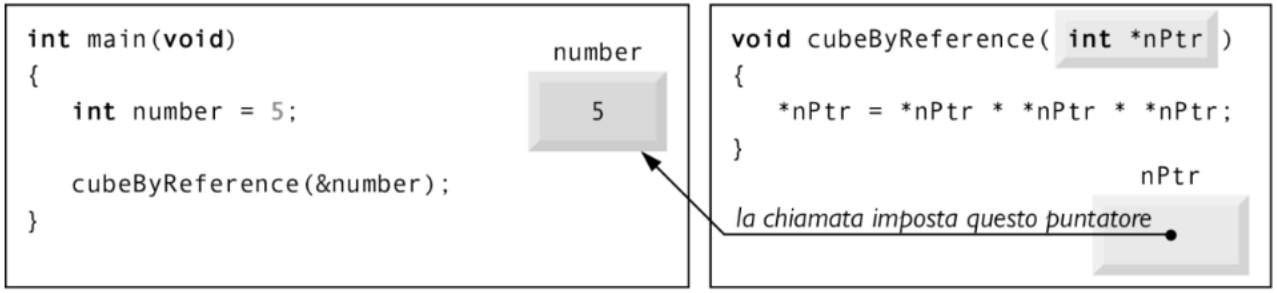
Fonte: Deitel & Deitel

Analisi di un tipico passaggio per riferimento

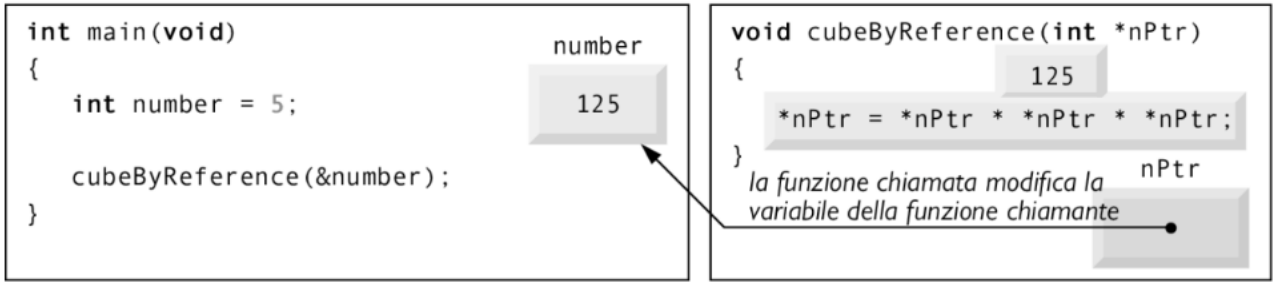
Passo 1: Prima che `main` chiami `cubeByReference`:



Passo 2: Dopo che `cubeByReference` ha ricevuto la chiamata e prima che `*nPtr` sia elevato al cubo:

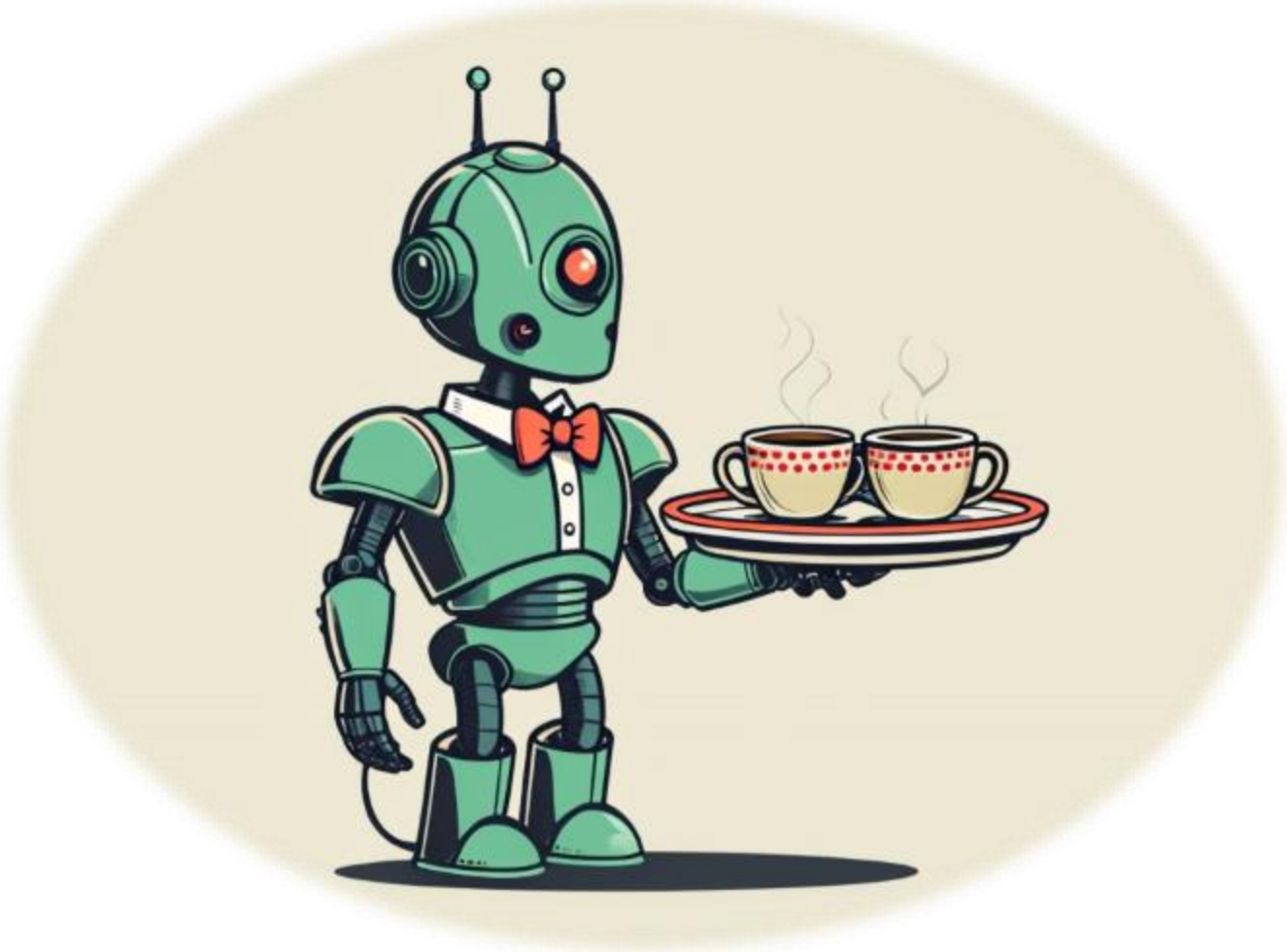


Passo 3: Dopo che `*nPtr` è stato elevato al cubo e prima che il controllo del programma torni alla funzione `main`:



Fonte: Deitel & Deitel

Intervallo



Fonte: PlaygroundAI

Bubble sort con passaggio per riferimento

- Deitel code example `fig07_11.c`
- La funzione `bubbleSort` ordina l'array e chiama la funzione `swap` per scambiare tra loro gli elementi `array[j]` e `array[j + 1]`
- Le funzioni non hanno accesso diretto agli elementi interni di altre funzioni per default
- La funzione `swap` non ha accesso diretto agli elementi dell'array di `bubbleSort`
- Anche se l'intero array è passato per riferimento, i singoli elementi (scalari) sono passati per valore
- Notate che l'intestazione della funzione bubbleSort dichiara array come int * const array invece che come int array[]
  - Queste notazioni sono intercambiabili; tuttavia, le notazioni con array sono di norma preferibili per una migliore leggibilità

Bubble sort con passaggio per riferimento

- `swap` riceve gli indirizzi degli elementi:
  - `element1Ptr` riceve `&array[j]`
  - `element2Ptr` riceve `&array[j + 1]`
- `*element1Ptr` è un sinonimo per `array[j]`, e `*element2Ptr` è un sinonimo per `array[j + 1]`
- Non si può usare direttamente:

```
int hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

- Ma si ottiene lo stesso effetto con:

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

Prototipo di una funzione nel corpo di un'altra funzione

- Il prototipo della funzione `swap` è inserito nel corpo della funzione `bubbleSort` perché `bubbleSort` è l'unica funzione che chiama `swap`
- Limita l'uso corretto di `swap` solo alle chiamate effettuate da `bubbleSort` o da funzioni che appaiono dopo di essa nel codice sorgente
- Funzioni definite prima di `swap` che tentano di chiamarla non hanno accesso al prototipo corretto.
- Il compilatore genera automaticamente un prototipo che spesso non corrisponde all'intestazione reale, causando avvisi o errori di compilazione
- Inserire i prototipi delle funzioni all'interno di altre funzioni limita le chiamate valide a quelle funzioni che includono esplicitamente i prototipi
- Riduce il rischio di utilizzi non autorizzati o indesiderati

Parametro `size` della funzione bubbleSort

- Necessario perché l'indirizzo del primo elemento dell'array non fornisce informazioni sul numero totale di elementi
- Si deve passare la dimensione dell'array come parametro per comunicare quanti elementi ordinare
- Alternativamente, si può passare un puntatore al primo elemento dell'array e un puntatore alla locazione subito oltre la fine dell'array; la differenza tra i due puntatori indica la lunghezza dell'array
- La funzione può essere usata per ordinare array di interi di qualsiasi dimensione

Parametro `size` della funzione bubbleSort

- Corretta applicazione dei principi di ingegneria del software
  - il passaggio della dimensione dell'array come parametro riduce la dipendenza da variabili globali
- Le variabili globali sono accessibili a tutto il programma, il che spesso viola il principio del privilegio minimo
- Le variabili globali dovrebbero essere usate solo per risorse veramente condivise (es. l'ora del giorno).
- La funzione sarebbe limitata all'elaborazione di array di una dimensione specifica, riducendo significativamente la sua riutilizzabilità.
- Solo i programmi che elaborano array interi unidimensionali di quella dimensione specifica potrebbero utilizzare la funzione

Operatore `sizeof`

- Utilizzato per determinare la dimensione in byte di un oggetto o di un tipo
- Applicato durante la fase di compilazione, a meno che l'operando non sia un array di lunghezza variabile (VLA)
- Quando applicato al nome di un array, `sizeof` restituisce il numero totale di byte nell'array (tipo `size_t`)
  - In un array di tipo `float` con 20 elementi ogni `float` occupa 4 byte
  - L'array occupa un totale di 80 byte
- `sizeof` è un operatore della fase di compilazione e non causa alcun sovraccarico di tempo di esecuzione (eccetto per i VLA)

Operatore `sizeof`

```c
// fig07_12.c
// L'applicazione di sizeof al nome di un array restituisce
// il numero di byte nell'array.
#include <stdio.h>
#define SIZE 20

size_t getSize( const float  *ptr ); // prototipo

int main(void){
   float array[SIZE]; // crea l'array

   printf("Number of bytes in the array is %zu\n", sizeof (array) );
    printf("Number of bytes returned by getSize is %zu\n", getSize(array) );
 }

 // restituisce la dimensione di ptr
 size_t getSize( const float  *ptr ) {
    return sizeof (ptr) ;
 }
```

```
Number of bytes in the array is 80
Number of bytes returned by getSize is 8
```

- Usare `sizeof` con un puntatore restituisce la *dimensione del puntatore*, non la dimensione dell'elemento al quale punta
- Sistemi a 64 bit: puntatore di size 8 bit

Fonte: Deitel & Deitel

Numero di elementi in un array

```
double real[22];
```

- Le variabili di tipo double occupano normalmente 8 byte in memoria
- L'array real contiene quindi 176 byte (22 elementi * 8 byte per elemento)

```
sizeof(real) / sizeof(real[0])
```

- Questa formula divide il numero totale di byte dell'array real per il numero di byte del suo primo elemento (un valore double)
- Questo calcolo funziona *solo* quando si usa il nome dell’array vero e proprio, *non* quando si usa un puntatore all’array

Determinare le dimensioni dei tipi standard, di un array e di un puntatore

```c
// fig07_13.c
// Uso dell'operatore sizeof per determinare le dimensioni di tipi standard.
#include <stdio.h>

int main(void) {
   char c = ;
   short s = 0;
   int i = 0;
   long l = 0;
   long long ll = 0;
    float f = 0.0F;
    double d = 0.0;
    long double ld = 0.0;
    int array[20] = {0}; // crea un array di 20 elementi int
    int *ptr = array; // crea un puntatore all'array

    printf("    sizeof c = %2zu\t       sizeof(char) = %2zu\n",
       sizeof c, sizeof(char));
    printf("    sizeof s = %2zu\t      sizeof(short) = %2zu\n",
       sizeof s, sizeof(short));
    printf("    sizeof i = %2zu\t        sizeof(int) = %2zu\n",
       sizeof i, sizeof(int));
    printf("    sizeof l = %2zu\t       sizeof(long) = %2zu\n",
       sizeof l, sizeof(long));
    printf("   sizeof ll = %2zu\t  sizeof(long long) = %2zu\n",
       sizeof ll, sizeof(long long));
    printf("    sizeof f = %2zu\t      sizeof(float) = %2zu\n",
       sizeof f, sizeof(float));
    printf("    sizeof d = %2zu\t     sizeof(double) = %2zu\n",
       sizeof d, sizeof(double));
    printf("   sizeof ld = %2zu\tsizeof(long double) = %2zu\n",
       sizeof ld, sizeof(long double));
    printf("sizeof array = %2zu\n  sizeof ptr = %2zu\n",
       sizeof array, sizeof ptr);
 }
```

```
    sizeof c =  1             sizeof(char) =  1
    sizeof s =  2            sizeof(short) =  2
    sizeof i =  4              sizeof(int) =  4
    sizeof l =  8             sizeof(long) =  8
   sizeof ll =  8        sizeof(long long) =  8
    sizeof f =  4            sizeof(float) =  4
    sizeof d =  8           sizeof(double) =  8
   sizeof ld = 16      sizeof(long double) = 16
sizeof array = 80
  sizeof ptr =  8
```

Fonte: Deitel & Deitel

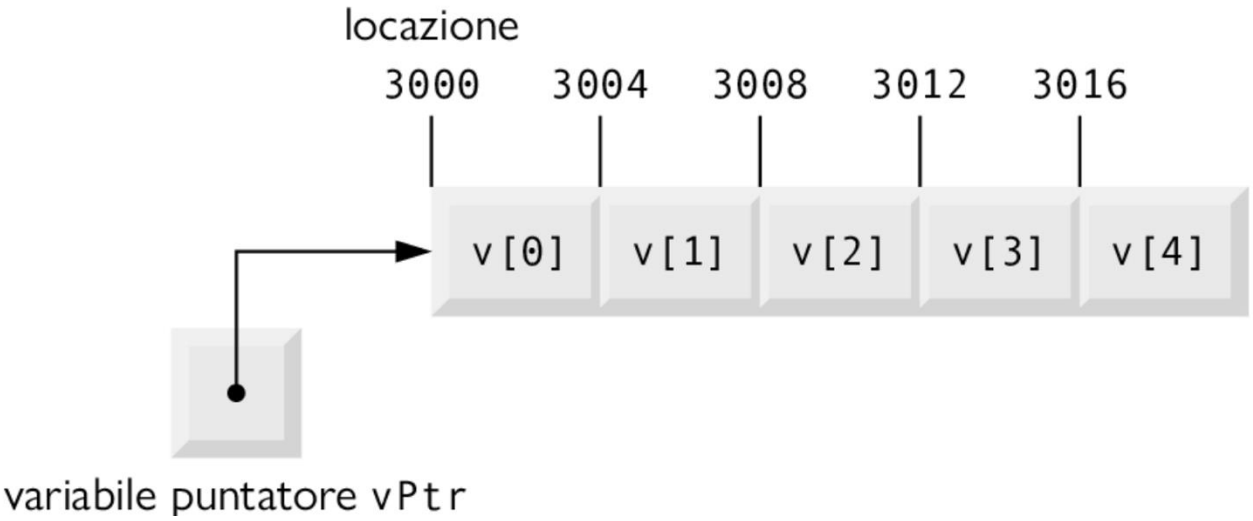
Determinare le dimensioni dei tipi standard, di un array e di un puntatore

- Il numero di byte utilizzati per memorizzare un tipo specifico può variare a seconda del sistema di computer
- Quando si scrivono programmi che dipendono dalle dimensioni dei tipi e devono funzionare su sistemi diversi, è consigliabile utilizzare `sizeof` per determinare il numero di byte usati per memorizzare i vari tipi
- `sizeof` può essere applicato a:
  - Un nome di variabile
  - Un tipo di dato
  - Un valore (incluso il valore di un'espressione)
- Quando `sizeof` è applicato al nome di una variabile (che non è il nome di un array) o a una costante, restituisce il numero di byte usati per memorizzare quel tipo specifico
- Le parentesi sono necessarie se l'operando di `sizeof` è un tipo di dato

Operatori per l'aritmetica dei puntatori

- È possibile eseguire le seguenti operazioni aritmetiche con i puntatori:
  - incrementare (++) o decrementare (--);
  - aggiungere un intero a un puntatore (+ o +=);
  - sottrarre un intero da un puntatore (- o -=);
  - sottrarre un puntatore da un altro puntatore (operazione che ha senso solo quando *entrambi* i puntatori puntano agli elementi dello *stesso* array)
- Usare l'aritmetica dei puntatori su puntatori che non fanno riferimento a elementi in un array è un errore logico

Indirizzare un puntatore a un array

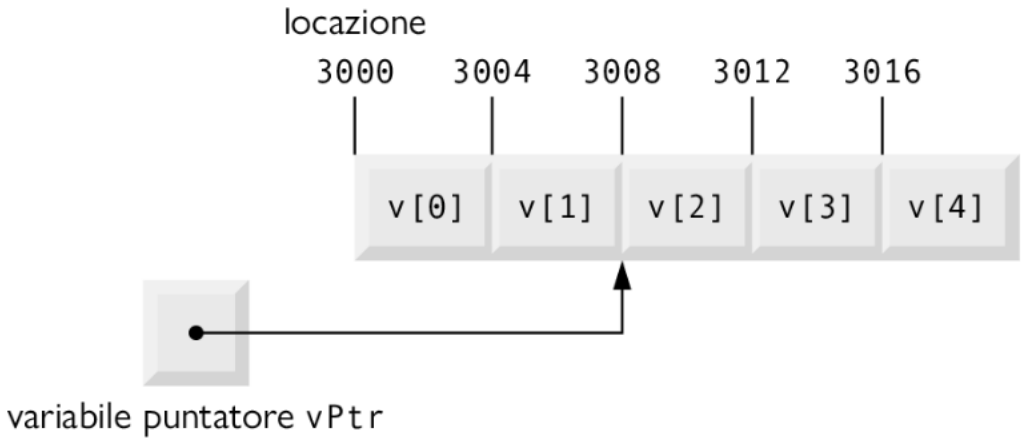


Fonte: Deitel & Deitel

- La variabile `vPtr` può essere inizializzata per puntare all'array `v` con l'una o l'altra delle istruzioni

```
vPtr = v;
vPtr = &v[0];
```

Aggiungere un intero a un puntatore



Fonte: Deitel & Deitel

- All'inizio il puntatore `vPtr` punta al primo elemento dell'array (`v[0]`), situato all'indirizzo 3000
- Quando eseguiamo l'operazione `vPtr += 2;` il puntatore viene incrementato di 2 volte la dimensione dell'oggetto a cui punta, che in questo caso è un `int` di 4 byte
  - Passa da 3000 a 3008 (=3000 + 2*4)
- Stessa cosa per la sottrazione di un intero

Sottrarre puntatori

- Se `vptr = 3000` e `vptr2 = 3008`, sono due puntatori che puntano a due elementi di un array allora

  `x = vptr2 - vptr`

  - Assegna a `x` il numero degli elementi dell'array compresi tra i due puntatori (x = 2 e non 8)
- L'aritmetica dei puntatori è indefinita, a meno che non sia eseguita su elementi dello stesso array
  - Non possiamo presumere che due variabili dello stesso tipo siano memorizzate in modo contiguo nella memoria, a meno che non siano elementi adiacenti di un array

Assegnare puntatori ad altri puntatori

- Un puntatore può essere assegnato a un altro puntatore solo se entrambi hanno lo stesso tipo
- Un puntatore a `void` (`void *`) è un puntatore generico che può rappresentare qualunque tipo di puntatore
  - A un puntatore di qualsiasi tipo (`int *`, `char *`, `float *`, ecc.) può essere assegnato un `void *`, e viceversa, senza bisogno di effettuare un cast esplicito
- Per un `void *`, il tipo di dato e la dimensione non sono noti dal compilatore, rendendo la dereferenziazione impossibile
  - Errore di sintassi

Confrontare puntatori

- È possibile confrontare i puntatori usando operatori di uguaglianza (==, !=) e relazionali (<, >, <=, >=).
- Tali confronti sono significativi solo se i puntatori puntano a elementi dello stesso array; altrimenti, si verificano **errori logici**
- I confronti tra puntatori comparano gli indirizzi di memoria a cui i puntatori fanno riferimento
  - Un confronto può indicare, ad esempio, se un puntatore punta a un elemento dell'array con un indice più alto rispetto a un altro
- Il confronto di puntatori è spesso utilizzato per verificare se un puntatore è `NULL`

Quiz

Relazione tra array e puntatori

- Array e puntatori sono strettamente correlati e spesso possono essere usati in modo intercambiabile
- Il nome di un array può essere considerato un puntatore costante al primo elemento dell'array
- I puntatori possono essere usati per eseguire qualsiasi operazione che coinvolga l'indicizzazione di un array

```
int b[5];
int *bPtr;
```

- il nome dell'array `b` (senza indice) è un puntatore al primo elemento dell'array
- Si può impostare `bPtr` uguale all'indirizzo del primo elemento dell'array `b` con l'istruzione `bPtr = b;` che è equivalente a scrivere `bPtr = &b[0];`

Notazione puntatore/offset

- L'elemento `b[3]` di un array può essere referenziato usando l'espressione con puntatori `*(bPtr + 3)`, dove `bPtr` è un puntatore che punta al primo elemento dell'array
- Il numero 3 nell'espressione rappresenta l'offset rispetto al puntatore `bPtr`, corrispondente all'indice dell'array
- Le parentesi sono necessarie per garantire che l'operazione di dereferenziazione `*` venga eseguita dopo l'operazione di offset `+`, poiché `*` ha una precedenza maggiore rispetto a `+`
- Un array può essere trattato come un puntatore e usato nell'aritmetica dei puntatori; ad esempio, `*(b + 3)` fa riferimento all'elemento `b[3]`
- Ogni espressione con array indicizzati può essere riscritta usando la notazione puntatore/offset, e l'utilizzo del nome dell'array come puntatore non ne modifica l'indirizzo

Notazione puntatore/indice

- I puntatori possono essere indicizzati come gli array
- Se `bPtr` ha il valore di `b` l'espressione `bPtr[1]`
  - si riferisce all'elemento dell'array `b[1]`
- l nome di un array punta sempre all'inizio dell'array, quindi è come un puntatore costante.
  - Pertanto, l'espressione `b += 3` è scorretta
  - tenta di modificare il valore del nome dell'array con l'aritmetica dei puntatori
- Tentare di modificare il valore del nome di un array con l'aritmetica dei puntatori genera un **errore di compilazione**

Esempi

- Deitel Code example `fig07_14.cpp`
  - usa i quattro metodi che abbiamo esaminato per il riferimento agli elementi di un array (indicizzazione di un array, puntatore/offset con il nome dell'array come puntatore, indicizzazione di un puntatore e puntatore/offset con un puntatore) per stampare i quattro elementi dell'array intero `b`
- Deitel Code example `fig07_15.c`
  - lustrare ulteriormente l'intercambiabilità fra array e puntatori, con due funzioni che copiano una stringa in un array di caratteri, ma sono implementate in maniera diversa

Array di puntatori

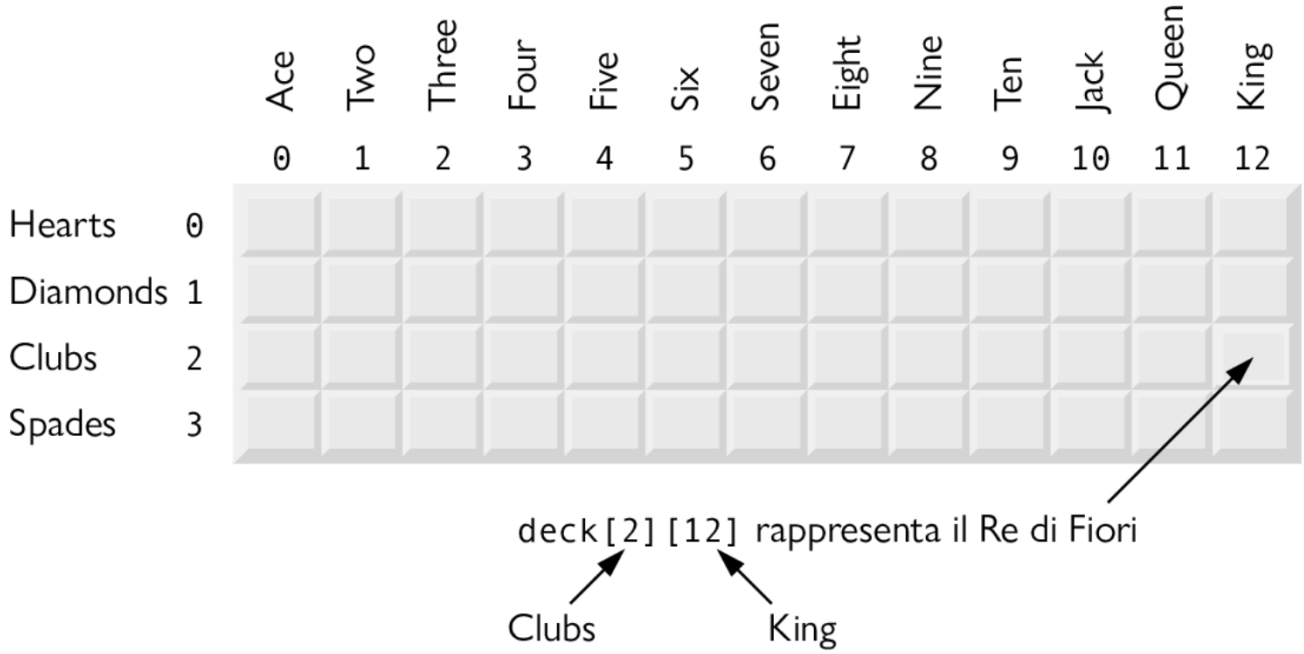
- Gli array possono contenere puntatori, e un uso comune è quello di creare un array di stringhe
- In C, ogni stringa è essenzialmente un puntatore al suo primo carattere
- Un array di stringhe è un array in cui ogni elemento è un puntatore al primo carattere di una stringa.

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"}
```

- L'array ha 4 elementi, ciascuno del tipo `char*` (puntatore a carattere)
- Le stringhe hanno lunghezze rispettive di 7, 9, 6 e 7 caratteri, tenendo conto del carattere nullo di terminazione.

Caso pratico di simulazione di numeri casuali: mescolare e distribuire le carte

- Deitel Code example `fig07_16.c`
- C'è un punto di debolezza nell'algoritmo per distribuire le carte



Fonte: Deitel & Deitel

Recap

- Definizioni e inizializzazione di variabili puntatore
- Operatori per i puntatori
- Passare argomenti a funzioni per riferimento
- Operatore sizeof
- Relazioni tra puntatori e array
- Array di puntatori