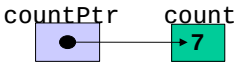
Capitolo 7 – I puntatori in C

7.2 Dichiarazione e inizializzazione di puntatori

- Variabili puntatore
  - Contengono gli indirizzi di memoria come valore
  - Le normali variabili contengono uno specifico valore (riferimento diretto)

  - I puntatori contengono gli indirizzi di una variabile che ha uno specifico valore (riferimento indiretto)
  - Referenziare – far riferimento al valore di un puntatore



7.2 Dichiarazione e inizializzazione di puntatori

- Dichiarazione di puntatori
  - ***** utilizzato con le variabili puntatore

```
int *myPtr;
```

  - Definisce un puntatore ad un `int` (puntatore di tipo `int *`)
  - Puntatori multipli richiedono l'uso di un * prima di ogni definizione di variabile

```
int *myPtr1, *myPtr2;
```

  - Si possono definire puntatori ad ogni tipo di dato
    - Il tipo del pointer dipende dal tipo della variabile a cui punta
    - Parliamo quindi di pointer a int, pointer a double, e così via

```
long k;
long *py = &k;    /*Sì*/
int *py = &k;     /*NO!*/
```

  - Inizializzare un puntatore a `0`, `NULL`, o ad un indirizzo
    - `0` o `NULL` – puntano a niente (`NULL` è preferito)

7.2 Dichiarazione e inizializzazione di puntatori

- E' lecito fare assegnazioni tra pointer dello stesso tipo:

```
int n;           /*normale variabile int*/
int *px = &n;    /*pointer a n*/
int *pz;         /*altro pointer a int*/

pz = px;         /*anche pz punta a n*/
```

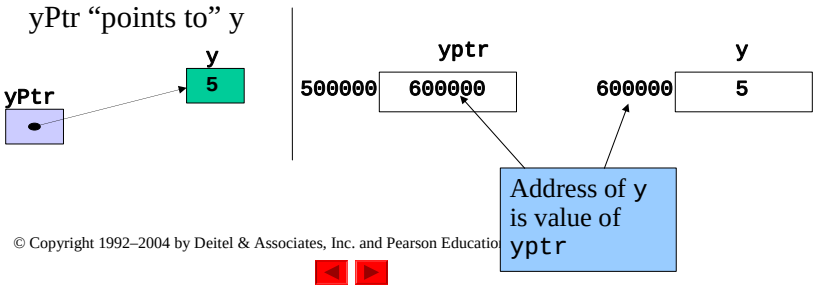
  - Ora anche pz contiene l'indirizzo di n, per cui si può accedere ad n attraverso pz:

```
*pz = 15;        /*come dire: n = 15*/
```


7.3 Operatori su puntatori

- & (operatore di indirizzo)
  - Restituisce l'indirizzo dell'operando (deve essere una variabile; & non si applica a costanti, espressioni o variabili register)

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
```



7.3 Operatori su puntatori

- * (*operatore di deriferimento o operatore di risoluzione del riferimento*)
  - Restituisce il valore dell'oggetto puntato dal suo operando
    - `*yptr` restituisce `y` (poiché `yptr` punta a `y`)

```
Printf("%d", *yPtr); /*stampa il valore di y*/
```

(risoluzione del riferimento di un puntatore)

  - * può essere utilizzato per l'assegnamento
    - Restituisce l'alias ad un oggetto

```
*yptr = 7;  /* changes y to 7 */
```

- * e & sono l'uno il complemento dell'altro

```c
/* Fig. 7.4: fig07_04.c
   Using the & and * operators */
#include <stdio.h>

int main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

L'indirizzo di a è il valore di aPtr.

L'operatore * restituisce un alias a cui l'operando punta. aPtr punta ad a, quindi *aPtr restituisce a.

Si noti come * e & sono inversi

Outline

fig07_04.c

7


```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Outline

Program Output

8

7.4 Chiamata di funzioni per riferimento

- La chiamata per riferimento con argomenti puntatore
  - Si passa l'indirizzo dell'argomento utilizzando l'operatore &
  - Permette di modificare la reale locazione di memoria
  - Gli array non sono passati con & poiché il nome di un array è già un puntatore
- operatore *
  - Utilizzato come alias/nickname per la variabile in una funzione

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - `*number` usato come nickname per la variabile passata

Outline

fig07_06.c

```
1  /* Fig. 7.6: fig07_06.c
2     Cube a variable using call-by-value */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* prototype */
6
7  int main()
8  {
9     int number = 5; /* initialize number */
10
11    printf( "The original value of number is %d", number );
12
13    /* pass number by value to cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "\nThe new value of number is %d\n", number );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
22 /* calculate and return cube of integer argument */
23 int cubeByValue( int n )
24 {
25    return n * n * n;   /* cube local variable n and return result */
26
27 } /* end function cubeByValue */
```

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

Outline

11

Program Output

Outline

12

fig07_07.c

```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference with a pointer argument */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /*
7
8  int main()
9  {
10    int number = 5; /* initialize number */
11
12    printf( "The original value of number is %d", number );
13
14    /* pass address of number to cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
23 /* calculate cube of *nPtr; modifies variable number in main */
24 void cubeByReference( int *nPtr )
25 {
26    *nPtr = *nPtr * *nPtr * *nPtr; /* cube *nPtr */
27 } /* end function cubeByReference */
```

Notice that the function prototype takes a pointer to an integer.

Notice how the address of `number` is given - `cubeByReference` expects a pointer (an address of a variable).

Inside `cubeByReference`, `*nPtr` is used (`*nPtr` is `number`).

Outline

Program Output

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


Before `main` calls `cubeByValue` :

```
int main()
{
   int number = 5;

   number=cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

After `cubeByValue` receives the call:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main` :

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

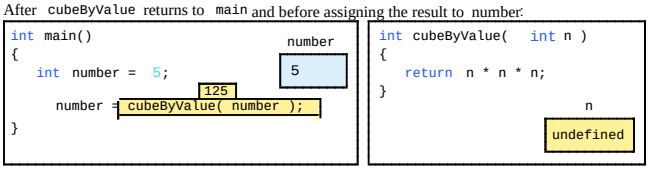
```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

Fig. 7.8 Analysis of a typical call-by-value. (Part 1 of 2.)

After `cubeByValue` returns to `main` and before assigning the result to `number`:



After `main` completes the assignment to `number`:

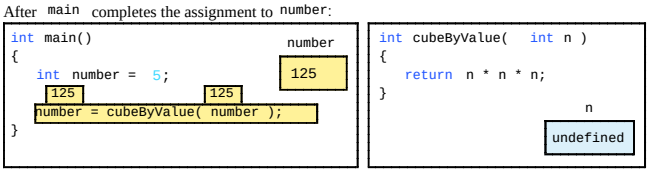
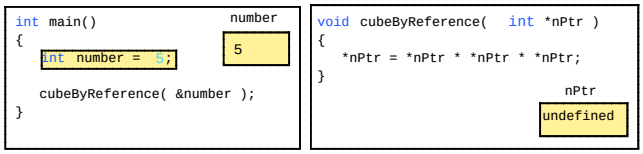
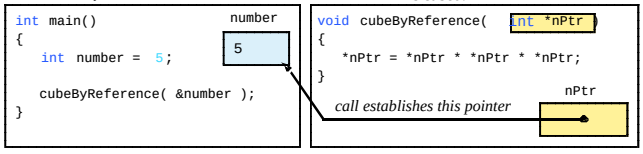


Fig. 7.8 Analysis of a typical call-by-value. (Part 2 of 2.)

Before `main` calls `cubeByReference`:



After `cubeByReference` receives the call and before `*nPtr` is cubed:



After `*nPtr` is cubed and before program control returns to `main`:

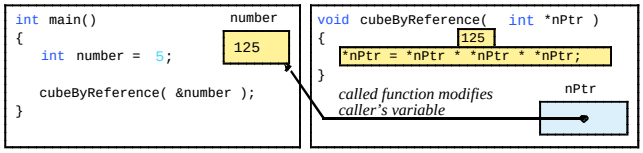


Fig. 7.9 Analysis of a typical call-by-reference with a pointer argument.

7.5 Usare il qualificatore `const` con i puntatori

- il qualificatore `const`
  - La varibile non può essere cambiata
  - Usare `const` se la funzione non necessita di modificare una variabile
  - Cercare di modificare una variabile `const` provoca un errore
- puntatori `const`
  - Puntano ad una locazione di memoria costante
  - Devono essere inizializzati quando definiti
  - `int *const myPtr = &x;`
    - Puntatore costante ad un `int`
  - `const int *myPtr = &x;`
    - Puntatore generico a un `const int`
  - `const int *const Ptr = &x;`
    - `const` punta a `const int`
    - `x` può essere cambiato, ma non `*Ptr`

Outline

fig07_10.c (Part 1 of 2)

```
1  /* Fig. 7.10: fig07_10.c
2     Converting lowercase letters to uppercase letters
3     using a non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototype */
9
10 int main()
11 {
12    char string[] = "characters and $32.98"; /* initialize char array */
13
14    printf( "The string before conversion is: %s", string );
15    convertToUppercase( string );
16    printf( "\nThe string after conversion is: %s\n", string );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

```
22 /* convert string to uppercase letters */
23 void convertToUppercase( char *sPtr )
24 {
25    while ( *sPtr != '\0' ) { /* current character is not '\0' */
26
27       if ( islower( *sPtr ) ) {    /* if character is lowercase, */
28          *sPtr = toupper( *sPtr ); /* convert to uppercase */
29       } /* end if */
30
31       ++sPtr;  /* move sPtr to the next character */
32    } /* end while */
33
34 } /* end function convertToUppercase */
```


Outline

fig07_10.c (Part 2 of 2)

Program Output

```
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```



```
1  /* Fig. 7.11: fig07_11.c
2     Printing a string one character at a time using
3     a non-constant pointer to constant data */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main()
10 {
11    /* initialize char array */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
15    printCharacters( string );
16    printf( "\n" );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```


Outline

fig07_11.c (Part 1 of 2)

```
22 /* sPtr cannot modify the character to which it points,
23    i.e., sPtr is a "read-only" pointer */
24 void printCharacters( const char *sPtr )
25 {
26    /* loop through entire string */
27    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
28       printf( "%c", *sPtr );
29    } /* end for */
30
31 } /* end function printCharacters */
```


Outline

fig07_11.c (Part 2 of 2)

Program Output

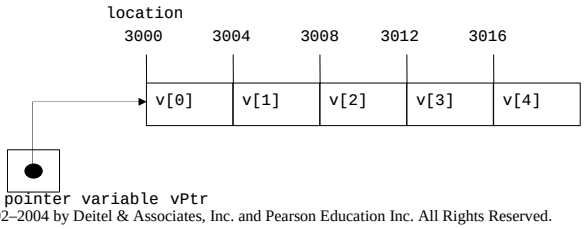
```
The string is:
print characters of a string
```


7.7 Espressioni puntatore e puntatori aritmetici

- Si possono effettuare delle operazioni aritmetiche sui puntatori
  - Incremento/decremento di puntatore (++ o --)
  - Aggiungere un intero ad un puntatore (+ o += , - o -=)
  - Un puntatore può essere sottratto da un altro
  - Si possono fare operazioni di confronto tra due puntatori

7.7 Espressioni puntatore e puntatori aritmetici

- 5 elementi di un array di `int` su una macchina a 4 byte `int`s
  - `vPtr` punta al primo elemento `v[ 0 ]`
    - Alla locazione `3000` (`vPtr = 3000`)
  - `vPtr += 2;` setta `vPtr` a `3008`
  - `vPtr` punta a `v[ 2 ]` (incrementato di 2), ma la macchina ha 4 byte `int`s, quindi punta all'indirizzo `3008`



7.7 Espressioni puntatore e puntatori aritmetici

- Sottrarre i puntatori
  - Restituisce il numero di elementi. Se

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
```

  - `vPtr2 - vPtr` dovrebbe produrre 2
- Confronto di puntatori (`<`, `==` , `>`)
  - Controlla quale puntatore punta all'elemento più alto dell'array

7.7 Espressioni puntatore e puntatori aritmetici

- Puntatori dello stesso tipo possono essere assegnati l'uno all'altro
  - Se non sono dello stesso tipo bisogna usare un operatore di cast
  - Eccezione: puntatore a `void` (type `void *`)
    - Puntatore generico, rappresenta qualsiasi tipo
    - Non c'è bisogno di casting per convertire un puntatore a un puntatore a `void`
    - I puntatori `void` non possono essere dereferenziati

Esercizi

- Esercizio 1

Scrivere una funzione con prototipo

void somma(double *sum, double x, double y);

  - che assegna la somma fra il secondo e il terzo argomento (passati per valore) al primo argomento (passato per riferimento)

- Esercizio 2

Dichiarare il vettore di tipo float chiamato numbers contenente 10 elementi e inizializzateli con i valori 0.0, 1.1, 2.2, ..., 9.9

  - dichiarare un puntatore nPtr che faccia riferimento a un oggetto di tipo float
  - visualizzare gli elementi del vettore utilizzando la notazione con gli indici di vettore
  - assegnare al puntatore nPtr l'indirizzo di partenza del vettore numbers
  - visualizzare gli elementi del vettore utilizzando la notazione con puntatore

- Esercizio 3

Scrivere una funzione che restituisca il puntatore alla prima occorrenza di un numero in un array di interi

int * cerca(int a[], int size)

- Esercizio 4

Scrivere una funzione che ha in input due array di interi (entrambi della stessa dimensione), ed un intero rappresentante la loro dimensione, e restituisce 1 se gli array sono uguali e 0 altrimenti. Utilizzare i puntatori per scorrere gli array.

int confronta(int a1[], int a2[], int size)