Capitolo 7 – I puntatori in C

Outline

Obiettivi

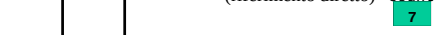
- In questo capitolo, impareremo a
  - Utilizzare i puntatori;
  - Utilizzare i puntatori per il passaggio di argomenti a funzioni utilizzando la chiamata per riferimento;
  - Capire la stretta relazione fra puntatori, array e stringhe;
  - Capire l'uso di puntatori a funzioni;
  - Definire ed utilizzare array di stringhe.

Introduzione

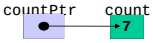
- Puntatori
  - Potenti, ma dificili da gestire
  - Simulazione del call-by-reference
  - Stretta relazione fra array e stringhe

Dichiarazione e inizializzazione di puntatori

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



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



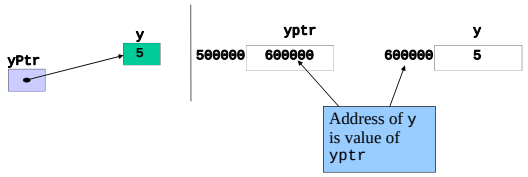
Dichiarazione e inizializzazione di puntatori

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

Operatori su puntatori

- & (operatore di indirizzo)
  - Restituisce l'indirizzo dell'operando

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



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)
  - * può essere utilizzato per l'assegnamento
    - Restituisce l'alias ad un oggetto
      ```
      *yptr = 7;  /* changes y to 7 */
      ```
  - puntatore dereferenziato (operando di *) deve essere un lvalue (non una costante)
- * e & sono l'uno il complemento dell'altro


```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;        /* a is an integer */
8     int *aPtr;    /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;    /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are complements of "
20            "each other\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
```

The address of a is the value of `aPtr`.

The * operator returns an alias to what its operand points to. `aPtr` points to a, so `*aPtr` returns a.

Notice how * and & are inverses

```
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

Outline

Operatori sui puntatori

Operators							Associativity	Type
( )	[ ]						left to right	highest
+	-	++	--	!	*	& (type)	right to left	unary
*	/	%					left to right	multiplicative
+	-						left to right	additive
<	<=	>	>=				left to right	relational
==	!=						left to right	equality
&&							left to right	logical and
||							left to right	logical or
?:							right to left	conditional
=	+=	-=	*=	/=	%=		right to left	assignment
,							left to right	comma

Fig. 7.5 Operator precedence.

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


```
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 */
```

Outline

fig07_06.c

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

Outline

Program Output


```
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 ); /* prototype */
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

fig07_07.c

© Copyright 1992–2004 by Deitel & Associates, Inc. and Pearson Education Inc. All Rights Reserved.

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

Outline

Program Output

© Copyright 1992–2004 by Deitel & Associates, Inc. and Pearson Education Inc. All Rights Reserved.

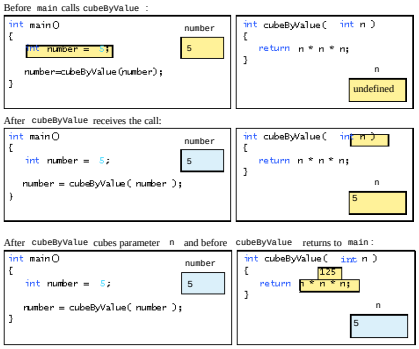
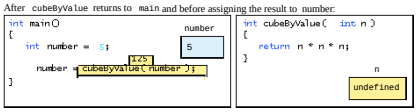


Fig. 7.8 Analysis of a typical call-by-value. (Part 1 of 2.)

© Copyright 1992–2004 by Deitel & Associates, Inc. and Pearson Education Inc. All Rights Reserved.

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



After main completes the assignment to number:

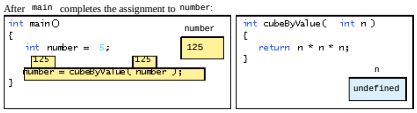
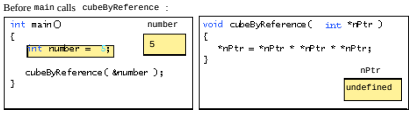
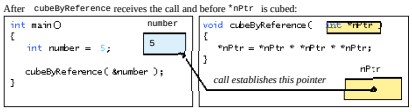


Fig. 7.8 Analysis of a typical call-by-value. (Part 2 of 2.)

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 :

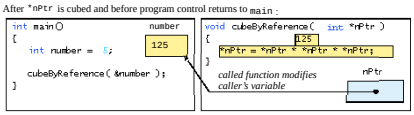


Fig. 7.9 Analysis of a typical call-by-reference with a pointer argument.

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;`
    - Type `int *const` – 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`


```
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
```

Outline

fig07_10.c (Part 1 of 2)

```
/* convert string to uppercase letters */
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) { /* current character is not '\0' */

      if ( islower( *sPtr ) ) {    /* if character is lowercase, */
         *sPtr = toupper( *sPtr ); /* convert to uppercase */
      } /* end if */

      ++sPtr;  /* move sPtr to the next character */
   } /* end while */

} /* 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
```



```
/* Fig. 7.11: fig07_11.c
   Printing a string one character at a time using
   a non-constant pointer to constant data */

#include <stdio.h>

void printCharacters( const char *sPtr );

int main()
{
   /* initialize char array */
   char string[] = "print characters of a string";

   printf( "The string is:\n" );
   printCharacters( string );
   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

Outline

fig07_11.c (Part 1 of 2)


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

} /* end function printCharacters */
```

Outline

fig07_11.c (Part 2 of 2)

Program Output

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



```
/* Fig. 7.12: fig07_12.c
   Attempting to modify data through a
   non-constant pointer to constant data. */
#include <stdio.h>

void f( const int *xPtr ); /* prototype */

int main()
{
   int y;       /* define y */

   f( &y );     /* f attempts illegal modification */

   return 0;    /* indicates successful termination */

} /* end main */

/* xPtr cannot be used to modify the
   value of the variable to which it points */
void f( const int *xPtr )
{
   *xPtr = 100;  /* error: cannot modify a const object */
} /* end function f */
```

Outline

fig07_12.c

```
Compiling...
FIG07_12.c
d:\books\2003\chtp4\examples\ch07\fig07_12.c(22) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

Outline

Program Output


```
1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to non-constant data */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x; /* define x */
8     int y; /* define y */
9
10    /* ptr is a constant pointer to an integer that can be modified
11       through ptr, but ptr always points to the same memory location */
12    int * const ptr = &x;
13
14    *ptr = 7; /* allowed: *ptr is not const */
15    ptr = &y; /* error: ptr is const; cannot assign new address */
16
17    return 0; /* indicates successful termination */
18
19 } /* end main */
```

Changing `*ptr` is allowed – `x` is not a constant.

Changing `ptr` is an error – `ptr` is a constant pointer.

```
Compiling...
FIG07_13.c
D:\books\2003\chtp4\Examples\ch07\FIG07_13.c(15) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

Outline

fig07_13.c

Program Output


```
1  /* Fig. 7.14: fig07_14.c
2     Attempting to modify a constant pointer to constant data. */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x = 5; /* initialize x */
8     int y;     /* define y */
9
10    /* ptr is a constant pointer to a constant integer. ptr always
11       points to the same location; the integer at that location
12       cannot be modified */
13    const int *const ptr = &x;
14
15    printf( "%d\n", *ptr );
16
17    *ptr = 7; /* error: *ptr is const; cannot assign new value */
18    ptr = &y; /* error: ptr is const; cannot assign new address */
19
20    return 0; /* indicates successful termination */
21
22 } /* end main */
```

Outline

fig07_14.c


```
Compiling...
FIG07_14.c
D:\books\2003\chtp4\Examples\ch07\FIG07_14.c(17) : error C2166: l-value
   specifies const object
D:\books\2003\chtp4\Examples\ch07\FIG07_14.c(18) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

Outline

Program Output

Bubble Sort utilizzando la chiamata per riferimento

- Implementazione del bubblesort utilizzando i puntatori
  - Scambio di due elementi
  - La funzione `swap` deve ricevere l'indirizzo (usando `&`) di un array di elementi
  - Usando i puntatori e l'operatore `*`, `swap` può scambiare elementi di un array
- Pseudo codice

  Initialize array
  print data in original order
  Call function bubblesort
  print sorted array
  Define bubblesort

Bubble Sort utilizzando la chiamata per riferimento

- `sizeof`
  - Restituisce la dimensione in byte dell'operando
  - Per gli array: size di 1 elemento * numero di elementi
  - se `sizeof( int )` è uguale a 4 byte, allora

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - stamperà `40`
- `sizeof` può essere usato con
  - Nomi di variabili
  - Tipi di nomi
  - Valori costanti


```
1  /* Fig. 7.15: fig07_15.c
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int *array, const int size ); /* prototype */
8
9  int main()
10 {
11    /* initialize array a */
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    int i; /* counter */
15
16    printf( "Data items in original order\n" );
17
18    /* loop through array a */
19    for ( i = 0; i < SIZE; i++ ) {
20       printf( "%4d", a[ i ] );
21    } /* end for */
22
23    bubbleSort( a, SIZE ); /* sort the array */
24
25    printf( "\nData items in ascending order\n" );
26
```

`Bubblesort` gets passed the address of array elements (pointers). The name of an array is a pointer.

Outline

fig07_15.c (Part 1 of 3)


```
27    /* loop through array a */
28    for ( i = 0; i < SIZE; i++ ) {
29       printf( "%4d", a[ i ] );
30    } /* end for */
31
32    printf( "\n" );
33
34    return 0; /* indicates successful termination */
35
36 } /* end main */
37
38 /* sort an array of integers using bubble sort algorithm */
39 void bubbleSort( int *array, const int size )
40 {
41    void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
42    int pass; /* pass counter */
43    int j;    /* comparison counter */
44
45    /* loop to control passes */
46    for ( pass = 0; pass < size - 1; pass++ ) {
47
48       /* loop to control comparisons during each pass */
49       for ( j = 0; j < size - 1; j++ ) {
50
```

Outline

fig07_15.c (Part 2 of 3)

```
         /* swap adjacent elements if they are out of order */
         if ( array[ j ] > array[ j + 1 ] ) {
            swap( &array[ j ], &array[ j + 1 ] );
         } /* end if */

      } /* end inner for */

   } /* end outer for */

} /* end function bubbleSort */

/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```


fig07_15.c (Part 3 of 3)

Program Output


```
/* Fig. 7.16: fig07_16.c
   Sizeof operator when used on an array name
   returns the number of bytes in the array. */
#include <stdio.h>

size_t getSize( float *ptr ); /* prototype */

int main()
{
   float array[ 20 ]; /* create array */

   printf( "The number of bytes in the array is %d"
           "\nThe number of bytes returned by getSize is %d\n",
           sizeof( array ), getSize( array ) );

   return 0; /* indicates successful termination */

} /* end main */

/* return size of ptr */
size_t getSize( float *ptr )
{
   return sizeof( ptr );

} /* end function getSize */
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```


fig07_16.c

Program Output


```
/* Fig. 7.17: fig07_17.c
   Demonstrating the sizeof operator */
#include <stdio.h>

int main()
{
   char c;          /* define c */
   short s;         /* define s */
   int i;           /* define i */
   long l;          /* define l */
   float f;         /* define f */
   double d;        /* define d */
   long double ld;  /* define ld */
   int array[ 20 ]; /* initialize array */
   int *ptr = array; /* create pointer to array */

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


fig07_17.c (Part 1 of 2)


```
           sizeof c, sizeof( char ), sizeof s,
           sizeof( short ), sizeof i, sizeof( int ),
           sizeof l, sizeof( long ), sizeof f,
           sizeof( float ), sizeof d, sizeof( double ),
           sizeof ld, sizeof( long double ),
           sizeof array, sizeof ptr );

   return 0; /* indicates successful termination */

} /* end main */
```

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


fig07_17.c (Part 2 of 2)

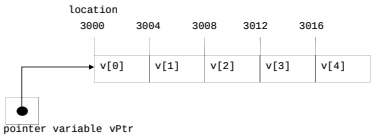
Program Output

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

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`



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
  - Controlla se un puntatore punta a **`0`**

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

Relazione fra puntatori ed array

- Gli array e i puntatori sono fortemente correlati
  - Nome di array come puntatore costante
  - I puntatori possono accedere agli elementi di un array
- Definire un array `b[ 5 ]` ed un puntatore `bPtr`
  - Per settarli uguali l'uno all'altro:

    ```
    bPtr = b;
    ```

    - Il nome dell'array (`b`) è in realtà l'indirizzo del primo elemento dell'array `b[ 5 ]`

      ```
      bPtr = &b[ 0 ]
      ```

    - Assegna in modo esplicito `bPtr` all'indirizzo del primo elemento di `b`

Relazione fra puntatori ed array

- L'elemento `b[ 3 ]`
  - si può accedere con `*( bPtr + 3 )`
    - dove n è l'. Detta pointer/offset notation
  - si può accedere con `bptr[ 3 ]`
    - Detta pointer/subscript notation
    - `bPtr[ 3 ]` come `b[ 3 ]`
  - si può accedere effettuando un'operazione aritmetica sull'operatore

    ```
    *( b + 3 )
    ```



```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
```

Outline

fig07_20.c (Part 1 of 2)


```
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```

Outline

fig07_20.c (Part 2 of 2)

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Outline

Program Output


```
1  /* Fig. 7.21: fig07_21.c
2     Copying a string using array notation and pointer notation. */
3  #include <stdio.h>
4
5  void copy1( char *s1, const char *s2 ); /* prototype */
6  void copy2( char *s1, const char *s2 ); /* prototype */
7
8  int main()
9  {
10    char string1[ 10 ];          /* create array string1 */
11    char *string2 = "Hello";     /* create a pointer to a string */
12    char string3[ 10 ];          /* create array string3 */
13    char string4[] = "Good Bye"; /* create a pointer to a string */
14
15    copy1( string1, string2 );
16    printf( "string1 = %s\n", string1 );
17
18    copy2( string3, string4 );
19    printf( "string3 = %s\n", string3 );
20
21    return 0; /* indicates successful termination */
22
23 } /* end main */
24
```

Outline

fig07_21.c (Part 1 of 2)


```
25 /* copy s2 to s1 using array notation */
26 void copy1( char *s1, const char *s2 )
27 {
28    int i; /* counter */
29
30    /* loop through strings */
31    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
32       ; /* do nothing in body */
33    } /* end for */
34
35 } /* end function copy1 */
36
37 /* copy s2 to s1 using pointer notation */
38 void copy2( char *s1, const char *s2 )
39 {
40    /* loop through strings */
41    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42       ; /* do nothing in body */
43    } /* end for */
44
45 } /* end function copy2 */
```

```
string1 = Hello
string3 = Good Bye
```

Outline

fig07_21.c (Part 2 of 2)

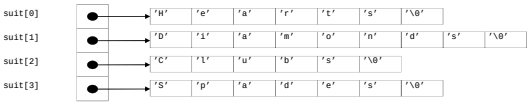
Program Output

Array di puntatori

- Gli arrays possono contenere puntatori
- Per esempio: un array di stringhe

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

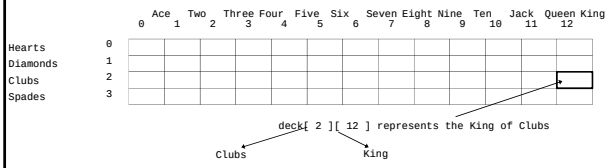
  - Le stringhe sono dei puntatori al primo carattere
  - `char *` – ogni elemento di `suit` è un puntatore a `char`
  - Le stringhe non sono in realtà memorizzare nell'array `suit`, sono memeorizzati solo i puntatori alle stringhe



  - l'array `suit` ha una dimensione fissa, ma le stringhe possono essere di qualsiasi dimensione

Simulazione di mescolatore e distributore di carte

- Card shuffling program
  - Use array of pointers to strings
  - Use double scripted array (suit, face)



  - The numbers 1-52 go into the array
    - Representing the order in which the cards are dealt

Simulazione di mescolatore e distributore di carte

- Pseudocode
  - Top level:
    Shuffle and deal 52 cards
  - First refinement:
    Initialize the suit array
    Initialize the face array
    Initialize the deck array
    Shuffle the deck
    Deal 52 cards

Simulazione di mescolatore e distributore di carte

  - Second refinement
    - Convert *shuffle the deck* to
      For each of the 52 cards
      Place card number in randomly selected unoccupied slot of deck
    - Convert *deal 52 cards* to
      For each of the 52 cards
      Find card number in deck array and print face and suit of card

Simulazione di mescolatore e distributore di carte

  - Third refinement
    - Convert *shuffle the deck* to
      Choose slot of deck randomly
      While chosen slot of deck has been previously chosen
      Choose slot of deck randomly
      Place card number in chosen slot of deck
    - Convert *deal 52 cards* to
      For each slot of the deck array
      If slot contains card number
      Print the face and suit of the card

```
1  /* Fig. 7.24: fig07_24.c
2     Card shuffling dealing program */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* prototypes */
8  void shuffle( int wDeck[][ 13 ] );
9  void deal( const int wDeck[][ 13 ], const char *wFace[],
10            const char *wSuit[] );
11
12 int main()
13 {
14    /* initialize suit array */
15    const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
16
17    /* initialize face array */
18    const char *face[ 13 ] =
19       { "Ace", "Deuce", "Three", "Four",
20         "Five", "Six", "Seven", "Eight",
21         "Nine", "Ten", "Jack", "Queen", "King" };
22
23    /* initialize deck array */
24    int deck[ 4 ][ 13 ] = { 0 };
25
```


Outline

fig07_24.c (Part 1 of 4)

```
25    srand( time( 0 ) ); /* seed random-number generator */
26
27    shuffle( deck );
28    deal( deck, face, suit );
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
33
34 /* shuffle cards in deck */
35 void shuffle( int wDeck[][ 13 ] )
36 {
37    int row;    /* row number */
38    int column; /* column number */
39    int card;   /* counter */
40
41    /* for each of the 52 cards, choose slot of deck randomly */
42    for ( card = 1; card <= 52; card++ ) {
43
44       /* choose new random location until unoccupied slot found */
45       do {
46          row = rand() % 4;
47          column = rand() % 13;
48       } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
49
50
```


Outline

fig07_24.c (Part 2 of 4)

```
51       /* place card number in chosen slot of deck */
52       wDeck[ row ][ column ] = card;
53    } /* end for */
54
55 } /* end function shuffle */
56
57 /* deal cards in deck */
58 void deal( const int wDeck[][ 13 ], const char *wFace[],
59           const char *wSuit[] )
60 {
61    int card;   /* card counter */
62    int row;    /* row counter */
63    int column; /* column counter */
64
65    /* deal each of the 52 cards */
66    for ( card = 1; card <= 52; card++ ) {
67
68       /* loop through rows of wDeck */
69       for ( row = 0; row <= 3; row++ ) {
70
71          /* loop through columns of wDeck for current row */
72          for ( column = 0; column <= 12; column++ ) {
73
74             /* if slot contains current card, display card */
75             if ( wDeck[ row ][ column ] == card ) {
```


Outline

fig07_24.c (Part 3 of 4)

```
76             printf( "%5s of %-8s", wFace[ column ], wSuit[ row ],
77                     card % 2 == 0 ? '\n' : '\t' );
78          } /* end if */
79
80       } /* end for */
81
82    } /* end for */
83
84 } /* end for */
85
86 } /* end function deal */
```


Outline

fig07_24.c (Part 4 of 4)

```
Nine of Hearts          Five of Clubs
Queen of Spades         Three of Spades
Queen of Hearts           Ace of Clubs
 King of Hearts           Six of Spades
 Jack of Diamonds        Five of Spades
Seven of Hearts          King of Clubs
Three of Clubs          Eight of Hearts
Three of Diamonds        Four of Diamonds
Queen of Diamonds        Five of Diamonds
  Six of Diamonds        Five of Hearts
   Ace of Spades           Six of Hearts
 Nine of Diamonds        Queen of Clubs
Eight of Spades          Nine of Clubs
Deuce of Clubs             Six of Clubs
Deuce of Spades          Jack of Clubs
 Four of Clubs           Eight of Clubs
 Four of Spades          Seven of Spades
Seven of Diamonds        Seven of Clubs
 King of Spades            Ten of Diamonds
 Jack of Hearts            Ace of Hearts
 Jack of Spades            Ten of Clubs
Eight of Diamonds        Deuce of Diamonds
  Ace of Diamonds          Nine of Spades
 Four of Hearts          Deuce of Hearts
 King of Diamonds          Ten of Spades
Three of Hearts            Ten of Hearts
```

Outline

Program Output

Puntatori a funzioni

- Puntatore a funzione
  - Contiene l'indirizzo della funzione
  - Simile a come il nome dell'array è l'indirizzo del primo elemento
  - Il nome della funzione è l'indirizzo iniziale del codice che definisce la funzione
- I puntatori a funzione possono essere
  - passati a funzioni
  - memorizzati in array
  - asseganti ad altri puntatori a funzioni

Puntatori a funzioni

- Esempio: bubblesort
  - La funzione `bubble` ha in input un puntatore a funzione
    - `bubble` chiama questa helper function
    - ciò determina l'ordinamento ascendente e discendente
  - L'argomento in `bubblesort` per il puntatore a funzione:

    ```
    int ( *compare )( int a, int b )
    ```

    dice a `bubblesort` di aspettarsi un puntatore ad una funzione che ha due `int`s come input e restituisce un `int`
  - Se non vengono usate le parentesi:

    ```
    int *compare( int a, int b )
    ```

    - Definisce una funzione che riceve due interi come input e restituisce un puntatore a `int`


```
1  /* Fig. 7.28: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* prototypes */
7  void bubble( int work[], const int size, int (*compare)( int a, int b ) );
8  int ascending( int a, int b );
9  int descending( int a, int b );
10
11 int main()
12 {
13    int order;   /* 1 for ascending order or 2 for descending order */
14    int counter; /* counter */
15
16    /* initialize array a */
17    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    printf( "Enter 1 to sort in ascending order,\n"
20            "Enter 2 to sort in descending order: " );
21    scanf( "%d", &order );
22
23    printf( "\nData items in original order\n" );
24
```

Outline

fig07_26.c (Part 1 of 4)

```
25    /* output original array */
26    for ( counter = 0; counter < SIZE; counter++ ) {
27       printf( "%5d", a[ counter ] );
28    } /* end for */
29
30    /* sort array in ascending order; pass function ascending as an
31       argument to specify ascending sorting order */
32    if ( order == 1 ) {
33       bubble( a, SIZE, ascending );
34       printf( "\nData items in ascending order\n" );
35    } /* end if */
36    else { /* pass function descending */
37       bubble( a, SIZE, descending );
38       printf( "\nData items in descending order\n" );
39    } /* end else */
40
41    /* output sorted array */
42    for ( counter = 0; counter < SIZE; counter++ ) {
43       printf( "%5d", a[ counter ] );
44    } /* end for */
45
46    printf( "\n" );
47
48    return 0; /* indicates successful termination */
49
50 } /* end main */
51
```


Outline

fig07_26.c (Part 2 of 4)

```
52 /* multipurpose bubble sort; parameter compare is a pointer to
53    the comparison function that determines sorting order */
54 void bubble( int work[], const int size, int (*compare)( int a, int b ) )
55 {
56    int pass;  /* pass counter */
57    int count; /* comparison counter */
58
59    void swap( int *element1Ptr, int *element2ptr ); /* prototype */
60
61    /* loop to control passes */
62    for ( pass = 1; pass < size; pass++ ) {
63
64       /* loop to control number of comparisons per pass */
65       for ( count = 0; count < size - 1; count++ ) {
66
67          /* if adjacent elements are out of order, swap them */
68          if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69             swap( &work[ count ], &work[ count + 1 ] );
70          } /* end if */
71
72       } /* end for */
73
74    } /* end for */
75
76 } /* end function bubble */
77
```


Outline

fig07_26.c (Part 3 of 4)

```
78 /* swap values at memory locations to which element1Ptr and
79    element2Ptr point */
80 void swap( int *element1Ptr, int *element2Ptr )
81 {
82    int hold; /* temporary holding variable */
83
84    hold = *element1Ptr;
85    *element1Ptr = *element2Ptr;
86    *element2Ptr = hold;
87 } /* end function swap */
88
89 /* determine whether elements are out of order for an ascending
90    order sort */
91 int ascending( int a, int b )
92 {
93    return b < a; /* swap if b is less than a */
94
95 } /* end function ascending */
96
97 /* determine whether elements are out of order for a descending
98    order sort */
99 int descending( int a, int b )
100 {
101    return b > a; /* swap if b is greater than a */
102
103 } /* end function descending */
```


Outline

fig07_26.c (Part 4 of 4)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```


Outline

Program Output