

## Capitolo 10 - Strutture, Unioni, Manipolazione di Bit, ed Enumerazioni

### Outline

- 10.1 Introduzione
- 10.2 Definizione di Strutture
- 10.3 Inizializzazione di strutture
- 10.4 Accesso alle componenti di una struttura
- 10.5 Utilizzo delle strutture con le funzioni
- 10.6 typedef
- 10.7 Esempio
- 10.8 Union
- 10.9 Operatori Bitwise
- 10.10 Campi di Bit
- 10.11 Enumerazioni

### Obiettivi

- In questo capitolo, impareremo a:
  - Creare e utilizzare le strutture, le union e le enumerazioni.
  - Passare strutture a funzioni: *call by value* e *call by reference*.
  - Manipolare i dati con gli operatori bitwise.
  - Creare i campi di bit per la memorizzazione compatta dei dati.

### 10.1 Introduzione

- Strutture
  - Collezioni di variabili correlate (aggregati) sotto un nome
    - Può contenere variabili di tipo diverso
  - Utilizzate di solito per definire record da memorizzare nei file
  - Combinare con i puntatori, possono essere utilizzate per la creazione di liste collegate, pile, code ed alberi

### 10.2 Definizione di strutture

- Esempio

```
struct card {
    char *face;
    char *suit;
};
```

  - `struct` introduce la definizione della struttura `card`
  - `card` è il nome della struttura ed è utilizzato per dichiarare variabili di questo tipo di struttura
  - `card` contiene due componenti di tipo `char *`
    - Queste componenti sono `face` e `suit`

## 10.2 Definizione di strutture

- **struct**
  - Una struct non può contenere un'istanza di se stessa
  - Non può contenere una componente che è un puntatore allo stesso tipo di struttura
  - Una definizione di struttura non alloca spazio in memoria
    - Crea invece un nuovo tipo di dato (astrazione dati)
- **Definizioni**
  - Definita come le altre variabili:

```
card oneCard, deck[ 52 ], *cPtr;
```
  - Si può utilizzare una lista separata da virgole:

```
struct card {  
    char *face;  
    char *suit;  
} oneCard, deck[ 52 ], *cPtr;
```

## 10.2 Definizione di strutture



Fig. 10.1) A possible storage alignment for a variable of type struct example showing an undefined area in memory.

## 10.2 Definizione di strutture

- **Operazioni valide**
  - Assegnare una struttura ad una struttura dello stesso tipo
  - Ottenere l'indirizzo (&) di una struttura
  - Accedere alle componenti di una struttura
  - Utilizzare l'operatore sizeof per determinare la dimensione della struttura

## 10.3 Inizializzazione di strutture

- **Liste di inizializzazione**
  - Esempio:

```
card oneCard = { "Three", "Hearts" };
```
- **Assegnamento**
  - Esempio:

```
card threeHearts = oneCard;
```
  - Si potrebbe definire ed inizializzare threeHearts come segue:

```
card threeHearts;  
threeHearts.face = "Three";  
threeHearts.suit = "Hearts";
```

## 10.4 Accedere alle componenti

- Accedere alle componenti di una struttura
  - L'operatore punto (.) utilizzato con la variabile struttura  
card myCard;  
printf( "%s", myCard.suit );
  - L'operatore freccia (->) utilizzato per puntatori a strutture  
card \*myCardPtr = &myCard;  
printf( "%s", myCardPtr->suit );
  - myCardPtr->suit è equivalente a  
( \*myCardPtr ).suit

```
1 /* Fig. 10.2: fig10_02.c
2 Using the structure member and
3 structure pointer operators */
4 #include <stdio.h>
5
6 /* card structure definition */
7 struct card {
8     char *face; /* define pointer face */
9     char *suit; /* define pointer suit */
10 }; /* end structure card */
11
12 int main()
13 {
14     struct card a; /* define struct a */
15     struct card *aPtr; /* define a pointer to card */
16
17     /* place strings into card structures */
18     a.face = "Ace";
19     a.suit = "Spades";
20
21     aPtr = &a; /* assign address of a to aPtr */
22 }
```

Outline  
fig10\_02.c (Part 1 of 2)

```
23 printf( "%s\n", myCardPtr->face, myCardPtr->suit );
24 aPtr->face = "of "; aPtr->suit;
25 ( *aPtr ).face = " of "; ( *aPtr ).suit );
26
27 return 0; /* indicates successful termination */
28
29 } /* end main */
```

Ace of Spades  
Ace of Spades  
Ace of Spades

Program Output

## 10.5 Utilizzare le strutture con le funzioni

- Passaggio di strutture a funzioni
  - Passare l'intera struttura
    - O, passare le singole componenti
  - Passaggio call by value
- Passare strutture call-by-reference
  - Passare il suo indirizzo
  - Passare il suo riferimento
- Passare array call-by-value
  - Creare una struttura con array come componente
  - Passare la struttura

## 10.6 typedef

- typedef
  - Crea dei sinonimi (alias) per tipi di dati definiti precedentemente
  - Utilizzare la typedef per creare tipi di nomi più corti
  - Esempio:

```
typedef struct Card *CardPtr;
```
  - Definisce un nuovo nome di tipo di dato CardPtr come sinonimo per il tipo struct Card \*
  - typedef non crea un nuovo tipo di dato
    - Crea solo un alias

## 10.7 Esempio

- Pseudo codice:
  - Creare un array di strutture card
  - Mettere le carte nel mazzo
  - Mescolare il mazzo
  - Distribuire le carte

```
1 /* Fig. 10.3: fig10_03.c
2 The card shuffling and dealing program using structures */
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 /* card structure definition */
8 struct card {
9     const char *face; /* define pointer face */
10    const char *suit; /* define pointer suit */
11 }; /* end structure card */
12
13 typedef struct card Card;
14
15 /* prototypes */
16 void fillDeck( Card * const wDeck, const char * wFace[]);
17 const char * wSuit[];
18 void shuffle( Card * const wDeck );
19 void deal( const Card * const wDeck );
20
21 int main()
22 {
23     Card deck[ 52 ]; /* define array of Cards */
24 }
```

Outline

fig10\_03.c (Part 1 of 4)

```
25 /* Initialize array of pointers */
26 const char *wFace[] = { "Ace", "Deuce", "Three", "Four", "Five",
27 "Six", "Seven", "Eight", "Nine", "Ten",
28 "Jack", "Queen", "King" };
29
30 /* Initialize array of pointers */
31 const char *wSuit[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
32
33 srand( time( NULL ) ); /* randomize */
34
35 fillDeck( deck, wFace, wSuit ); /* load the deck with Cards */
36 shuffle( deck ); /* put Cards in random order */
37 deal( deck ); /* deal all 52 Cards */
38
39 return 0; /* indicates successful termination */
40 } /* end main */
41
42 /* place strings into Card structures */
43 void fillDeck( Card * const wDeck, const char * wFace[],
44 const char * wSuit[] )
45 {
46     int i; /* counter */
47
48 }
```

Outline

fig10\_03.c (Part 2 of 4)

```

59 /* loop through wDeck */
60 for ( i = 0; i <= 51; i++ ) {
61     wDeck[ i ].face = wFace[ i % 13 ];
62     wDeck[ i ].suit = wsuit[ i / 13 ];
63 } /* end for */
64 } /* end function fillDeck */
65
66 /* shuffle cards */
67 void shuffle( Card * const wDeck )
68 {
69     int i; /* counter */
70     int j; /* variable to hold random value between 0 - 51 */
71     Card temp; /* define temporary structure for swapping Cards */
72
73     /* loop through wDeck randomly swapping Cards */
74     for ( i = 0; i <= 51; i++ ) {
75         j = rand() % 52;
76         temp = wDeck[ i ];
77         wDeck[ i ] = wDeck[ j ];
78         wDeck[ j ] = temp;
79     } /* end for */
80 } /* end function shuffle */

```

Outline  
fig10\_03.c (3 of 4)

```

74 /* deal cards */
75 void deal( const Card * const wDeck )
76 {
77     int i; /* counter */
78
79     /* loop through wDeck */
80     for ( i = 0; i <= 51; i++ ) {
81         printf( "005 of %s-0606", wDeck[ i ].face, wDeck[ i ].suit,
82             ( i + 1 ) % 2 ? "\n" : "\n" );
83     } /* end for */
84
85 } /* end function deal */

```

Outline  
fig10\_03.c (4 of 4)

```

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

```

Outline  
Program Output

## 10.8 Union

- **union**
  - Spazio di memoria che può contenere vari oggetti nel tempo
  - Contiene solo un dato alla volta
  - Le componenti di una union condividono spazio
  - Risparmia la memoria
  - Solo l'ultima componente definita è accessibile
- **union definizioni**
  - Come la struct

```

union Number {
    int x;
    float y;
};
union Number value;

```



```

1 /* Fig. 10.7: fig10_07.c
2  * Printing an unsigned integer in bits */
3 #include <stdio.h>
4
5 void displayBits( unsigned value ); /* prototype */
6
7 int main()
8 {
9     unsigned x; /* variable to hold user input */
10
11     printf( "Enter an unsigned integer: " );
12     scanf( "%u", &x );
13     displayBits( x );
14
15     return 0; /* indicates successful termination */
16 } /* end main */
17
18 /* display bits of an unsigned integer value */
19 void displayBits( unsigned value )
20 {
21     unsigned c; /* counter */
22
23
24

```

Outline  
fig10\_07.c (1 of 2)

```

25 /* define displayMask and left shift 31 bits */
26 unsigned displayMask = 1 << 31;
27
28 printf( "%u = ", value );
29
30 /* loop through bits */
31 for( c = 1; c <= 32; c++ ) {
32     putchar( value & displayMask ? '1' : '0' );
33     value <<= 1; /* shift value left by 1 */
34
35     if ( c % 8 == 0 ) { /* output space after 8 bits */
36         putchar( ' ' );
37     } /* end if */
38 } /* end for */
39
40 putchar( '\n' );
41 } /* end function displayBits */

```

Outline  
fig10\_07.c (2 of 2)

```

Enter an unsigned integer: 65000
65000 = 00000000 00000000 11111101 11101000

```

## 10.9 Operatori Bitwise

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 10.8 Results of combining two bits with the bitwise AND operator.

```

1 /* Fig. 10.9: fig10_09.c
2  * Using the bitwise AND, bitwise inclusive OR, bitwise
3  * exclusive OR and bitwise complement operators */
4 #include <stdio.h>
5
6 void displayBits( unsigned value ); /* prototype */
7
8 int main()
9 {
10     unsigned number1; /* define number1 */
11     unsigned number2; /* define number2 */
12     unsigned mask; /* define mask */
13     unsigned setBits; /* define setBits */
14
15     /* demonstrate bitwise */
16     number1 = 65535;
17     mask = 1;
18     printf( "The result of combining the following\n" );
19     displayBits( number1 );
20     displayBits( mask );
21     printf( "using the bitwise AND operator & is\n" );
22     displayBits( number1 & mask );
23

```

Outline  
fig10\_09.c (1 of 4)

```

24 /* demonstrate bitwise | */
25 number1 = 15;
26 setBits = 241;
27 printf("\nThe result of combining the following\n");
28 displayBits( number1 );
29 displayBits( setBits );
30 printf("using the bitwise inclusive OR operator | is\n");
31 displayBits( number1 | setBits );
32
33 /* demonstrate bitwise exclusive OR */
34 number1 = 139;
35 number2 = 199;
36 printf("\nThe result of combining the following\n");
37 displayBits( number1 );
38 displayBits( number2 );
39 printf("using the bitwise exclusive OR operator ^ is\n");
40 displayBits( number1 ^ number2 );
41
42 /* demonstrate bitwise complement */
43 number1 = 21845;
44 printf("\nThe one's complement of\n");
45 displayBits( number1 );
46 printf("is\n");
47 displayBits( -number1 );
48

```

Outline  
fig10\_09.c (2 of 4)

```

49 return 0; /* indicates successful termination */
50 } /* end main */
51
52 /* display bits of an unsigned integer value */
53 void displayBits( unsigned value )
54 {
55     unsigned c; /* counter */
56
57     /* declare displayMask and left shift 31 bits */
58     unsigned displayMask = 1 << 31;
59
60     printf( "%10u = ", value );
61
62     /* loop through bits */
63     for ( c = 1; c <= 32; c++ ) {
64         putchar( value & displayMask ? '1' : '0' );
65         value <<= 1; /* shift value left by 1 */
66
67         if ( c % 8 == 0 ) { /* output a space after 8 bits */
68             putchar( ' ' );
69         } /* end if */
70     } /* end for */
71 } /* end for */
72
73

```

Outline  
fig10\_09.c (3 of 4)

```

74 putchar( '\n' );
75 } /* end function displayBits */

```

Outline  
fig10\_09.c (4 of 4)  
Program Output

```

The result of combining the following
65535 = 00000000 00000000 11111111 11111111
1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
1 = 00000000 00000000 00000000 00000001

The result of combining the following
15 = 00000000 00000000 00000000 00001111
241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
255 = 00000000 00000000 00000000 11111111

The result of combining the following
139 = 00000000 00000000 00000000 10001011
199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
76 = 00000000 00000000 00000000 01001100

The one's complement of
21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010

```

## 10.9 Operatori Bitwise

Bit 1	Bit 2	Bit 1   Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 10.11 Results of combining two bits with the bitwise inclusive OR operator |.



## 10.9 Operatori Bitwise

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 10.12 Results of combining two bits with the bitwise exclusive OR operator ^.

33

```

1 /* Fig. 10.13: fig10_13.c
2 Using the bitwise shift operators */
3 #include <stdio.h>
4
5 void displayBits(unsigned value); /* prototype */
6
7 int main()
8 {
9     unsigned number1 = 960; /* initialize number1 */
10
11     /* demonstrate bitwise left shift */
12     printf("The result of left shifting\n");
13     displayBits(number1);
14     printf("8 bit positions using the ");
15     printf("left shift operator << is\n");
16     displayBits(number1 << 8);
17
18     /* demonstrate bitwise right shift */
19     printf("The result of right shifting\n");
20     displayBits(number1);
21     printf("8 bit positions using the ");
22     printf("right shift operator >> is\n");
23     displayBits(number1 >> 8);
24

```

Outline  
fig10\_13.c (1 of 2)

34

```

25     return 0; /* indicates successful verification */
26 } /* end main */
27
28 /* display bits of an unsigned integer value */
29 void displayBits(unsigned value)
30 {
31     unsigned c; /* counter */
32
33     /* declare displayMask and left shift 31 bits */
34     unsigned displayMask = 1 << 31;
35
36     printf("%3u = ", value);
37
38     /* loop through bits */
39     for (c = 1; c <= 32; c++) {
40         putchar(' ');
41         displayMask = displayMask >> 1;
42         value <<= 1; /* shift value left by 1 */
43         if (c % 8 == 0) { /* output a space after 8 bits */
44             putchar(' ');
45         } /* end if */
46     } /* end for */
47
48     putchar('\n');
49 } /* end function displayBits */

```

Outline  
fig10\_13.c (2 of 2)

35

```

The result of left shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
245760 = 00000000 00000011 11000000 00000000

The result of right shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
3 = 00000000 00000000 00000000 00000011

```

Outline  
Program Output

36

## 10.9 Operatori Bitwise

Bitwise assignment operators	
&=	Bitwise AND assignment operator.
=	Bitwise inclusive OR assignment operator.
^=	Bitwise exclusive OR assignment operator.
<<=	Left-shift assignment operator.
>>=	Right-shift assignment operator.

Fig. 10.14 The bitwise assignment operators.

## 10.9 Operatori Bitwise

Operator	Associativity	Type
() [] - --	left to right	Highest
+ - ++ -- ! & * - sizeof (type)	right to left	Unary
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	shifting
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	bitwise AND
^	left to right	bitwise OR
	left to right	bitwise OR
&&	left to right	logical AND
	left to right	logical OR
?:	right to left	conditional
= += -= *= /= &=  = ^= <<= >>= %=	right to left	assignment
,	left to right	comma

Fig. 10.15 Operator precedence and associativity.

## 10.10 Campi di Bit

- Campi di bit
  - Le componenti di una struttura le cui dimensioni (in bit) sono state specificate
  - Permette un miglior utilizzo della memoria
  - Devono essere definiti come `int` o `unsigned`
  - Non è possibile accedere ai singoli bit
- Definire i campi di bit
  - Segue una componente `unsigned` o `int` con un due punti (:) ed una costante intera che rappresenta la dimensione del campo
  - Esempio:
 

```
struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

## 10.10 Campi di Bit

- Campi di bit senza nome
  - Campi utilizzati come padding in una struttura
  - Niente si può memorizzare nei bit
 

```
struct Example {
    unsigned a : 13;
    unsigned : 3;
    unsigned b : 4;
};
```
  - I campi di bit senza nome con dimensione zero allineano il successivo campo di bit ad una nuova unità di memorizzazione

```

1 /* Fig. 10.16: fig10_16.c
2 Representing cards with bit fields in a struct */
3
4 #include <stdio.h>
5
6 /* bitCard structure definition with bit fields */
7 struct bitCard {
8     unsigned face : 4; /* 4 bits; 0-15 */
9     unsigned suit : 2; /* 2 bits; 0-3 */
10    unsigned color : 1; /* 1 bit; 0-1 */
11 }; /* end struct bitCard */
12
13 typedef struct bitCard Card;
14
15 void fillDeck( Card *const wDeck ); /* prototype */
16 void deal( const Card *const wDeck ); /* prototype */
17
18 int main()
19 {
20     Card deck[ 52 ]; /* create array of Cards */
21     fillDeck( deck );
22     deal( deck );
23     return 0; /* indicates successful termination */
24 }
25
26

```

41

Outline

fig10\_16.c (1 of 3)

```

27 } /* end main */
28
29 /* initialize Cards */
30 void fillDeck( Card *const wDeck )
31 {
32     int i; /* counter */
33
34     /* loop through wDeck */
35     for ( i = 0; i <= 51; i++ ) {
36         wDeck[ i ].face = i % 13;
37         wDeck[ i ].suit = i / 13;
38         wDeck[ i ].color = i / 26;
39     } /* end for */
40 } /* end function fillDeck */
41
42
43 /* output cards in two column format; cards 0-25 subscripted with
44 k1 (column 1); cards 26-51 subscripted k2 (column 2) */
45 void deal( const Card *const wDeck )
46 {
47     int k1; /* subscripts 0-25 */
48     int k2; /* subscripts 26-51 */
49

```

42

Outline

fig10\_16.c (2 of 3)

```

50 /* loop through wDeck */
51 for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
52     printf( "Card: %3d Suit: %3d Color: %3d\n",
53            wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
54     printf( "Card: %3d Suit: %3d Color: %3d\n",
55            wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
56 } /* end for */
57 } /* end function deal */
58

```

43

Outline

fig10\_16.c (3 of 3)

```

Card: 0 Suit: 0 Color: 0 Card: 0 Suit: 2 Color: 1
Card: 1 Suit: 0 Color: 0 Card: 1 Suit: 2 Color: 1
Card: 2 Suit: 0 Color: 0 Card: 2 Suit: 2 Color: 1
Card: 3 Suit: 0 Color: 0 Card: 3 Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0 Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0 Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0 Card: 6 Suit: 2 Color: 1
Card: 7 Suit: 0 Color: 0 Card: 7 Suit: 2 Color: 1
Card: 8 Suit: 0 Color: 0 Card: 8 Suit: 2 Color: 1
Card: 9 Suit: 0 Color: 0 Card: 9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0 Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0 Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0 Card: 12 Suit: 2 Color: 1
Card: 0 Suit: 1 Color: 0 Card: 0 Suit: 3 Color: 1
Card: 1 Suit: 1 Color: 0 Card: 1 Suit: 3 Color: 1
Card: 2 Suit: 1 Color: 0 Card: 2 Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0 Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0 Card: 4 Suit: 3 Color: 1
Card: 5 Suit: 1 Color: 0 Card: 5 Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0 Card: 6 Suit: 3 Color: 1
Card: 7 Suit: 1 Color: 0 Card: 7 Suit: 3 Color: 1
Card: 8 Suit: 1 Color: 0 Card: 8 Suit: 3 Color: 1
Card: 9 Suit: 1 Color: 0 Card: 9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0 Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0 Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0 Card: 12 Suit: 3 Color: 1

```

44

Outline

Program Output

## 10.11 Le costanti di enumerazione

### • Enumerazioni

- Insieme di costanti intere rappresentate da identificatori
- Le costanti di enumerazione sono come le costanti simboliche i cui valori vengono settati automaticamente
  - I valori partono da 0 e sono incrementati di 1
  - I valori possono essere assegnati esplicitamente con =
  - Necessitano di nomi unici
- Esempio:

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
AUG, SEP, OCT, NOV, DEC};
```

  - Crea un nuovo tipo enum Months in cui gli identificatori sono settati agli interi da 1 a 12
- Le variabili enumerazione possono assumere solo i valori della loro enumerazione

45

```
1 /* Fig. 10.18: fig10_18.c
2 Using an enumeration type */
3 #include <stdio.h>
4
5 /* enumeration constants represent months of the year */
6 enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
7             JUL, AUG, SEP, OCT, NOV, DEC };
8
9 int main()
10 {
11     enum months month; /* can contain any of the 12 months */
12
13     /* Initialize array of pointers */
14     const char *monthName[] = { "", "January", "February", "March",
15                               "April", "May", "June", "July", "August", "September", "October",
16                               "November", "December" };
17
18     /* Loop through months */
19     for ( month = JAN; month <= DEC; month++ ) {
20         printf( "%2d\t%s\n", month, monthName[ month ] );
21     } /* end for */
22
23     return 0; /* indicates successful termination */
24 } /* end main */
```

Outline  
fig10\_18.c

46

```
1 January
2 February
3 March
4 April
5 May
6 June
7 July
8 August
9 September
10 October
11 November
12 December
```

Outline  
Program Output

47