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

Outline

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 varibili di tipo diverso
  - Utilizzate di solito per definire record da memorizzare nei file
  - Combinate 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 puo contenere una componente che è un puntatore allo stesso tipo di struttura
  - Un definizione di struttura non alloca spazio in memoria
    - Crea invece un nuovo tipo di dato (astrazine 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

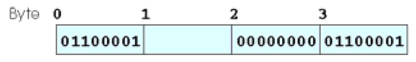

Fig. 10.1) A possible storage alignment for a variable of type `struct example` showing an undefined area in memory.§

10.2 Definizione di strutture

- Operazioni valide
  - Assegnare una struttura ad una strutura 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`

```c
/* Fig. 10.2: fig10_02.c
   Using the structure member and
   structure pointer operators */
#include <stdio.h>

/* card structure definition */
struct card {
   char *face; /* define pointer face */
   char *suit; /* define pointer suit */
}; /* end structure card */

int main()
{
   struct card a;     /* define struct a */
   struct card *aPtr; /* define a pointer to card */

   /* place strings into card structures */
   a.face = "Ace";
   a.suit = "Spades";

   aPtr = &a; /* assign address of a to aPtr */

```

Outline

fig10_02.c (Part 1 of 2)

```c
   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", a.face, " of ", a.suit,
           aPtr->face, " of ", aPtr->suit,
           ( *aPtr ).face, " of ", ( *aPtr ).suit );

   return 0; /* indicates successful termination */

} /* end main */
```

Outline

fig10_02.c (Part 2 of 2)

```
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

```
/* Fig. 10.3: fig10_03.c
   The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* card structure definition */
struct card {
   const char *face; /* define pointer face */
   const char *suit; /* define pointer suit */
}; /* end structure card */

typedef struct card Card;

/* prototypes */
void fillDeck( Card * const wDeck, const char * wFace[],
   const char * wSuit[] );
void shuffle( Card * const wDeck );
void deal( const Card * const wDeck );

int main()
{
   Card deck[ 52 ]; /* define array of Cards */

```

Outline

fig10_03.c (Part 1 of 4)

```
   /* initialize array of pointers */
   const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

   /* initialize array of pointers */
   const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

   srand( time( NULL ) ); /* randomize */

   fillDeck( deck, face, suit );/* load the deck with Cards */
   shuffle( deck ); /* put Cards in random order */
   deal( deck ); /* deal all 52 Cards */

   return 0; /* indicates successful termination */

} /* end main */

/* place strings into Card structures */
void fillDeck( Card * const wDeck, const char * wFace[],
   const char * wSuit[] )
{
   int i; /* counter */

```

Outline

fig10_03.c (Part 2 of 4)

```
   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   } /* end for */

} /* end function fillDeck */

/* shuffle cards */
void shuffle( Card * const wDeck )
{
   int i;     /* counter */
   int j;     /* variable to hold random value between 0 - 51 */
   Card temp; /* define temporary structure for swapping Cards */

   /* loop through wDeck randomly swapping Cards */
   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wDeck[ i ];
      wDeck[ i ] = wDeck[ j ];
      wDeck[ j ] = temp;
   } /* end for */

} /* end function shuffle */

```

Outline

fig10_03.c (3 of 4)

```
/* deal cards */
void deal( const Card * const wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      printf( "%5s of %-8s%c", wDeck[ i ].face, wDeck[ i ].suit,
         ( i + 1 ) % 2 ? '\t' : '\n' );
   } /* end for */

} /* 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;
```

10.8 Union

- Operazioni `union` valide
  - Assegnamento alla `union` dello stesso tipo: `=`
  - Ottenere l'indirizzo: `&`
  - Far riferimento alle componenti di una union: `.`
  - Far riferimento alle componenti utilizzando i puntatori: `->`

Outline

fig10_05.c (1 of 2)

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

/* number union definition */
union number {
   int x;     /* define int x */
   double y; /* define double y */
}; /* end union number */

int main()
{
   union number value; /* define union value */

   value.x = 100; /* put an integer into the union */
   printf( "%s\n%s\n%s%d\n%s%f\n\n",
          "Put a value in the integer member",
          "and print both members.",
          "int:    ", value.x,
          "double:\n", value.y );

```

Outline

fig10_05.c (2 of 2)

```c
   value.y = 100.0; /* put a double into the same union */
   printf( "%s\n%s\n%s%d\n%s%f\n",
          "Put a value in the floating member",
          "and print both members.",
          "int:    ", value.x,
          "double:\n", value.y );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Put a value in the integer member
and print both members.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000.000000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

10.9 Operatori Bitwise

- Tutti i dati sono rappresentati come sequenze di bit
  - Ogni bit può essere `0` o `1`
  - Sequenze di 8 bit costituiscono un byte

Operator		Description
`&`	bitwise AND	The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`.
`|`	bitwise inclusive OR	The bits in the result are set to `1` if at least one of the corresponding bits in the two operands is `1`.
`^`	bitwise exclusive OR	The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with `0` bits.
`>>`	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
`~`	one's complement	All `0` bits are set to `1` and all `1` bits are set to `0`.

Fig. 10.6 The bitwise operators.

Outline

fig10_07.c (1 of 2)

```c
/* Fig. 10.7: fig10_07.c
   Printing an unsigned integer in bits */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main()
{
   unsigned x; /* variable to hold user input */

   printf( "Enter an unsigned integer: " );
   scanf( "%u", &x );

   displayBits( x );

   return 0; /* indicates successful termination */

} /* end main */

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

```

Outline

fig10_07.c (2 of 2)

```c
   /* define displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%7u = ", value );

   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

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

```
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&.

Outline

fig10_09.c (1 of 4)

```c
/* Fig. 10.9: fig10_09.c
   Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR and bitwise complement operators */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main()
{
   unsigned number1; /* define number1 */
   unsigned number2; /* define number2 */
   unsigned mask;    /* define mask */
   unsigned setBits; /* define setBits */

   /* demonstrate bitwise & */
   number1 = 65535;
   mask = 1;
   printf( "The result of combining the following\n");
   displayBits( number1 );
   displayBits( mask );
   printf( "using the bitwise AND operator & is\n");
   displayBits( number1 & mask );

   /* demonstrate bitwise | */
   number1 = 15;
   setBits = 241;
   printf( "\nThe result of combining the following\n");
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator | is\n");
   displayBits( number1 | setBits );

   /* demonstrate bitwise exclusive OR */
   number1 = 139;
   number2 = 199;
   printf( "\nThe result of combining the following\n");
   displayBits( number1 );
   displayBits( number2 );
   printf( "using the bitwise exclusive OR operator ^ is\n");
   displayBits( number1 ^ number2 );

   /* demonstrate bitwise complement */
   number1 = 21845;
   printf( "\nThe one's complement of\n" );
   displayBits( number1 );
   printf( "is\n" );
   displayBits( ~number1 );

```

Outline

fig10_09.c (2 of 4)

```c
   return 0; /* indicates successful termination */

} /* end main */

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* declare displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ?'1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output a space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

```

Outline

fig10_09.c (3 of 4)

```c
   putchar( '\n' );
} /* end function displayBits */
```

```
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
```

Outline

fig10_09.c (4 of 4)

Program Output

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 ^.

Outline

fig10_13.c (1 of 2)

```c
/* Fig. 10.13: fig10_13.c
   Using the bitwise shift operators */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main()
{
   unsigned number1 = 960; /* initialize number1 */

    /* demonstrate bitwise left shift */
    printf( "\nThe result of left shifting\n" );
    displayBits( number1 );
    printf( "8 bit positions using the " );
    printf( "left shift operator << is\n" );
    displayBits( number1 << 8 );

    /* demonstrate bitwise right shift */
    printf( "\nThe result of right shifting\n" );
    displayBits( number1 );
    printf( "8 bit positions using the " );
    printf( "right shift operator >> is\n" );
    displayBits( number1 >> 8 );

```

Outline

fig10_13.c (2 of 2)

```c
   return 0; /* indicates successful termination */

} /* end main */

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* declare displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%7u = ", value );

   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output a space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

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

Outline

Program Output

```
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
```

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

Outline

fig10_16.c (1 of 3)

```c
/* Fig. 10.16: fig10_16.c
   Representing cards with bit fields in a struct */

#include <stdio.h>

/* bitCard structure definition with bit fields */
struct bitCard {
   unsigned face : 4;  /* 4 bits; 0-15 */
   unsigned suit : 2;  /* 2 bits; 0-3 */
   unsigned color : 1; /* 1 bit; 0-1 */
}; /* end struct bitCard */

typedef struct bitCard Card;

void fillDeck( Card * const wDeck );   /* prototype */
void deal( const Card * const wDeck ); /* prototype */

int main()
{
   Card deck[ 52 ]; /* create array of Cards */

   fillDeck( deck );
   deal( deck );

   return 0; /* indicates successful termination */

```

Outline

fig10_16.c (2 of 3)

```c
} /* end main */

/* initialize Cards */
void fillDeck( Card * const wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = i % 13;
      wDeck[ i ].suit = i / 13;
      wDeck[ i ].color = i / 26;
   } /* end for */

} /* end function fillDeck */

/* output cards in two column format; cards 0-25 subscripted with
   k1 (column 1); cards 26-51 subscripted k2 (column 2) */
void deal( const Card * const wDeck )
{
   int k1; /* subscripts 0-25 */
   int k2; /* subscripts 26-51 */

```

Outline

fig10_16.c (3 of 3)

```c
   /* loop through wDeck */
   for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      printf( "Card:%3d  Suit:%2d  Color:%2d   ",
         wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
      printf( "Card:%3d  Suit:%2d  Color:%2d\n",
         wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
   } /* end for */

} /* end function deal */
```

Outline

Program Output

```
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
```

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 assegant 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

fig10_18.c

```c
/* Fig. 10.18: fig10_18.c
   Using an enumeration type */
#include <stdio.h>

/* enumeration constants represent months of the year */
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };

int main()
{
   enum months month; /* can contain any of the 12 months */

   /* initialize array of pointers */
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   /* loop through months */
   for ( month = JAN; month <= DEC; month++ ) {
      printf( "%2d%11s\n", month, monthName[ month ] );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

Program Output

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