10

C Structures, Unions, Bit Manipulations and Enumerations

© 2007 Pearson Education, Inc. All rights reserved.

But yet an union in partition.

—William Shakespeare

The same old charitable lie
Repeated as the years scoot by
Perpetually makes a hit—
"You really haven't changed a bit!"

—Margaret Fishback

I could never make out what those damned dots meant.

—Winston Churchill

© 2007 Pearson Education, Inc. All rights reserved.

OBJECTIVES

In this chapter you will learn:

- To create and use structures, unions and enumerations.
- To pass structures to functions by value and by reference.
- To manipulate data with the bitwise operators.
- To create bit fields for storing data compactly.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

10.1 Introduction
10.2 Structure Definitions
10.3 Initializing Structures
10.4 Accessing Members of Structures
10.5 Using Structures with Functions
10.6 `typedef`
10.7 Example: High-Performance Card Shuffling and Dealing Simulation
10.8 Unions
10.9 Bitwise Operators
10.10 Bit Fields
10.11 Enumeration Constants

© 2007 Pearson Education, Inc. All rights reserved.

10.1 Introduction

- **Structures**
  - **Collections of related variables (aggregates) under one name**
    - **Can contain variables of different data types**
  - **Commonly used to define records to be stored in files**
  - **Combined with pointers, can create linked lists, stacks, queues, and trees**

© 2007 Pearson Education, Inc. All rights reserved.

10.2 Structure Definitions

- Example

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

  - struct introduces the definition for structure card
  - card is the structure name and is used to declare variables of the structure type
  - card contains two members of type char *
    - These members are face and suit

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.1

Forgetting the semicolon that terminates a structure definition is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

10.2 Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables
- Definitions
  - Defined like other variables:

```
card oneCard, deck[ 52 ], *cPtr;
```

  - Can use a comma separated list:

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

© 2007 Pearson Education, Inc. All rights reserved.

Good Programming Practice 10.1

Always provide a structure tag name when creating a structure type. The structure tag name is convenient for declaring new variables of the structure type later in the program.

© 2007 Pearson Education, Inc. All rights reserved.

Good Programming Practice 10.2

Choosing a meaningful structure tag name helps make a program self-documenting.

© 2007 Pearson Education, Inc. All rights reserved.

10.2 Structure Definitions

- Valid Operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the `sizeof` operator to determine the size of a structure

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.2

Assigning a structure of one type to a structure of a different type is a compilation error.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.3

Comparing structures is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

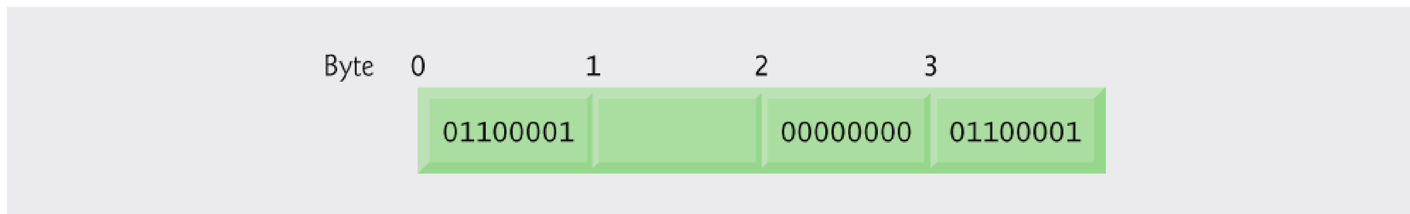

Fig. 10.1 | Possible storage alignment for a variable of type **`struct`** example showing an undefined area in memory.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.1

Because the size of data items of a particular type is machine dependent and because storage alignment considerations are machine dependent, so too is the representation of a structure.

© 2007 Pearson Education, Inc. All rights reserved.

10.3 Initializing Structures

- **Initializer lists**
  - Example:

```
card oneCard = { "Three", "Hearts" };
```

- **Assignment statements**
  - Example:

```
card threeHearts = oneCard;
```

  - Could also define and initialize `threeHearts` as follows:

```
card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

© 2007 Pearson Education, Inc. All rights reserved.

10.4 Accessing Members of Structures

- Accessing structure members
  - Dot operator (.) used with structure variables

```
card myCard;
printf( "%s", myCard.suit );
```

  - Arrow operator (->) used with pointers to structure variables

```
card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

  - `myCardPtr->suit` is equivalent to

```
( *myCardPtr ).suit
```

© 2007 Pearson Education, Inc. All rights reserved.

Error-Prevention Tip 10.1

Avoid using the same names for members of structures of different types. This is allowed, but it may cause confusion.

© 2007 Pearson Education, Inc. All rights reserved.

Good Programming Practice 10.3

Do not put spaces around the - > and . operators. Omitting spaces helps emphasize that the expressions the operators are contained in are essentially single variable names.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.4

Inserting space between the - and > components of the structure pointer operator (or between the components of any other multiple keystroke operator except ?:) is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.5

Attempting to refer to a member of a structure by using only the member's name is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.6

Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., `*cardPtr.suit`) is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
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( void )
13 {
14    struct card aCard; /* define one struct card variable */
15    struct card *cardPtr; /* define a pointer to a struct card */
16
17    /* place strings into aCard */
18    aCard.face = "Ace";
19    aCard.suit = "Spades";
```

Structure definition

Structure definition must end with semicolon

Dot operator accesses members of a structure

fig10_02.c

(1 of 2)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
20
21    cardPtr = &aCard; /* assign address of aCard to cardPtr */
22
23    printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
24       cardPtr->face, " of ", cardPtr->suit,
25       ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
26
27    return 0; /* indicates successful termination */
28
29 } /* end main */
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Arrow operator accesses members of a structure pointer

fig10_02.c

(2 of 2)

© 2007 Pearson Education, Inc. All rights reserved.

10.5 Using Structures with Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.7

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 10.1

Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

© 2007 Pearson Education, Inc. All rights reserved.

10.6 typedef

- `typedef`
  - Creates synonyms (aliases) for previously defined data types
  - Use `typedef` to create shorter type names
  - Example:

```
typedef struct Card *CardPtr;
```

  - Defines a new type name `CardPtr` as a synonym for type `struct Card *`
  - `typedef` does not create a new data type
    - Only creates an alias

© 2007 Pearson Education, Inc. All rights reserved.

Good Programming Practice 10.4

Capitalize the first letter of `typedef` names to emphasize that they are synonyms for other type names.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.2

Use `typedef` to help make a program more portable.

© 2007 Pearson Education, Inc. All rights reserved.

10.7 Example: High-Performance Card Shuffling and Dealing Simulation

- **Pseudocode:**
  - Create an array of card structures
  - Put cards in the deck
  - Shuffle the deck
  - Deal the cards

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_03.c

(1 of 3)

```
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; /* new type name for struct 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( void )
22 {
23    Card deck[ 52 ]; /* define array of Cards */
24
25    /* initialize array of pointers */
26    const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
27       "Six", "Seven", "Eight", "Nine", "Ten",
28       "Jack", "Queen", "King"};
29
```

Each **card** has a face and a suit

Card is now an alias for **struct card**

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
30    /* initialize array of pointers */
31    const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
32
33    srand( time( NULL ) ); /* randomize */
34
35    fillDeck( deck, face, suit ); /* 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
41 } /* end main */
42
43 /* place strings into Card structures */
44 void fillDeck( Card * const wDeck, const char * wFace[],
45    const char * wSuit[] )
46 {
47    int i; /* counter */
48
49    /* loop through wDeck */
50    for ( i = 0; i <= 51; i++ ) {
51       wDeck[ i ].face = wFace[ i % 13 ];
52       wDeck[ i ].suit = wSuit[ i / 13 ];
53    } /* end for */
54
55 } /* end function fillDeck */
56
```

Constant pointer to modifiable array of **Card**s

Fills the deck by giving each **Card** a face and suit

fig10_03.c

(2 of 3)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_03.c

(3 of 3)

```
57 /* shuffle cards */
58 void shuffle( Card * const wDeck )
59 {
60    int i;      /* counter */
61    int j;      /* variable to hold random value between 0 - 51 */
62    Card temp; /* define temporary structure for swapping Cards */
63
64    /* loop through wDeck randomly swapping Cards */
65    for ( i = 0; i <= 51; i++ ) {
66       j = rand() % 52;
67       temp = wDeck[ i ];
68       wDeck[ i ] = wDeck[ j ];
69       wDeck[ j ] = temp;
70    } /* end for */
71
72 } /* end function shuffle */
73
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( "%5s of %-8s%c", wDeck[ i ].face, wDeck[ i ].suit,
82          ( i + 1 ) % 2 ? '\t' : '\n' );
83    } /* end for */
84
85 } /* end function deal */
```

Each card is swapped with another, random card, shuffling the deck

© 2007 Pearson Education, Inc. All rights reserved.

```
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
```

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.8

Forgetting to include the array subscript when referring to individual structures in an array of structures is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

10.8 Unions

- union
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share space
  - Conserves storage
  - Only the last data member defined can be accessed
- union definitions
  - Same as struct

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

© 2007 Pearson Education, Inc. All rights reserved.

10.8 Unions

- **Valid uni on operations**
  - **Assignment to uni on of same type: =**
  - **Taking address: &**
  - **Accessing union members: .**
  - **Accessing members using pointers: - >**

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.9

Referencing data in a union with a variable of the wrong type is a logic error.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.3

If data is stored in a union as one type and referenced as another type, the results are implementation dependent.

© 2007 Pearson Education, Inc. All rights reserved.

Software Engineering Observation 10.1

As with a `struct` definition, a `union` definition simply creates a new type. Placing a `union` or `struct` definition outside any function does not create a global variable.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.10

Comparing unions is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.4

The amount of storage required to store a union is implementation dependent but will always be at least as large as the largest member of the union.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.5

Some unions may not port easily to other computer systems. Whether a union is portable or not often depends on the storage alignment requirements for the union member data types on a given system.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 10.2

Unions conserve storage.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
1  /* Fig. 10.5: fig10_05.c
2     An example of a union */
3  #include <stdio.h>
4
5  /* number union definition */
6  union number {
7     int x;
8     double y;
9  }; /* end union number */
10
11 int main( void )
12 {
13    union number value; /* define union variable */
14
15    value.x = 100; /* put an integer into the union */
16    printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n\n\n",
17       "Put a value in the integer member",
18       "and print both members.",
19       "int:", value.x,
20       "double:", value.y );
21
```

Union definition

Union definition must end with semicolon

Note that **y** has no value

fig10_05.c

(1 of 2)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
22     value.y = 100.0; /* put a double into the same union */
23     printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
24        "Put a value in the floating member",
25        "and print both members.",
26        "int:", value.x,
27        "double:", value.y );
28
29     return 0; /* indicates successful termination */
30
31  } /* end main */
```

Giving **y** a value removes **x**'s value

fig10_05.c

(2 of 2)

```
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
```

© 2007 Pearson Education, Inc. All rights reserved.

10.9 Bitwise Operators

- **All data is represented internally as sequences of bits**
  - **Each bit can be either 0 or 1**
  - **Sequence of 8 bits forms a byte**

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.6

Bitwise data manipulations are machine dependent.

© 2007 Pearson Education, Inc. All rights reserved.

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 | Bitwise operators.

© 2007 Pearson Education, Inc. All rights reserved.

```
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( void )
8  {
9     unsigned x; /* variable to hold user input */
10
11    printf( "Enter an unsigned integer: " );
12    scanf( "%u", &x );
13
14    displayBits( x );
15
16    return 0; /* indicates successful termination */
17
18 } /* end main */
19
```

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_07.c

(2 of 2)

```
20 /* display bits of an unsigned integer value */
21 void displayBits( unsigned value )
22 {
23    unsigned c; /* counter */
24
25    /* define displayMask and left shift 31 bits */
26    unsigned displayMask = 1 << 31;
27
28    printf( "%10u = ", 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
39    } /* end for */
40
41    putchar( '\n' );
42 } /* end function displayBits */
```

displayMask is a 1 followed by 31 zeros

Bitwise AND returns nonzero if the leftmost bits of **displayMask** and **value** are both 1, since all other bits in **displayMask** are 0s.

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

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.11

Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa is an error.

© 2007 Pearson Education, Inc. All rights reserved.

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 &.

© 2007 Pearson Education, Inc. All rights reserved.

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 `|`.

© 2007 Pearson Education, Inc. All rights reserved.

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 ^.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
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( void )
9  {
10    unsigned number1;
11    unsigned number2;
12    unsigned mask;
13    unsigned setBits;
14
15    /* demonstrate bitwise AND (&) */
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
```

Bitwise AND sets each bit in the result to 1 if the corresponding bits in the operands are both 1

fig10_09.c

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_09.c

(2 of 3)

```
24    /* demonstrate bitwise inclusive OR (|) */
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
49    return 0; /* indicates successful termination */
50 } /* end main */
51
```

Bitwise inclusive OR sets each bit in the result to 1 if at least one of the corresponding bits in the operands is 1

Bitwise exclusive OR sets each bit in the result to 1 if only one of the corresponding bits in the operands is 1

Complement operator sets each bit in the result to 0 if the corresponding bit in the operand is 1 and vice versa

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
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
71    } /* end for */
72
73    putchar( '\n' );
74 } /* end function displayBits */
```

fig10_09.c

(3 of 3)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
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
```

fig10_10.c

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.12

Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is an error.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_13.c

(1 of 3)

```
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( void )
8  {
9     unsigned number1 = 960; /* initialize number1 */
10
11    /* demonstrate bitwise left shift */
12    printf( "\nThe 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
```

Left shift operator shifts all bits left a specified number of spaces, filling in zeros for the empty bits

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
18    /* demonstrate bitwise right shift */
19    printf( "\nThe 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
25    return 0; /* indicates successful termination */
26 } /* end main */
27
28 /* display bits of an unsigned integer value */
29 void displayBits( unsigned value )
30 {
30 {
31    unsigned c; /* counter */
32
33    /* declare displayMask and left shift 31 bits */
34    unsigned displayMask = 1 << 31;
35
36    printf( "%7u = ", value );
37
```

fig10_13.c

(2 of 3)

Right shift operator shifts all bits right a specified number of spaces, filling in the empty bits in an implementation-defined manner

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_13.c

(3 of 3)

```
38    /* loop through bits */
39    for ( c = 1; c <= 32; c++ ) {
40       putchar( value & displayMask ? '1' : '0' );
41       value <<= 1; /* shift value left by 1 */
42
43       if ( c % 8 == 0 ) { /* output a space after 8 bits */
44          putchar( ' ' );
45       } /* end if */
46
47    } /* end for */
48
49    putchar( '\n' );
50 } /* end function displayBits */
```

```
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
```

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.13

The result of shifting a value is undefined if the right operand is negative or if the right operand is larger than the number of bits in which the left operand is stored.

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.7

Right shifting is machine dependent. Right shifting a signed integer fills the vacated bits with 0s on some machines and with 1s on others.

© 2007 Pearson Education, Inc. All rights reserved.

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.

© 2007 Pearson Education, Inc. All rights reserved.

Operators	Associativity	Type
() [ ] . ->	left to right	highest
+ - ++ -- ! & * ~ sizeof (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	shiting
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	bitwise AND

Fig. 10.15 | Operator precedence and associativity. (Part 1 of 2.)

© 2007 Pearson Education, Inc. All rights reserved.

<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>^</td><td>left to right</td><td>bitwise OR</td></tr>
<tr><td>|</td><td>left to right</td><td>bitwise OR</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= &= |= ^= <<= >>= %=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

Fig. 10.15 | Operator precedence and associativity. (Part 2 of 2.)

© 2007 Pearson Education, Inc. All rights reserved.

10.10 Bit Fields

- Bit field
  - Member of a structure whose size (in bits) has been specified
  - Enable better memory utilization
  - Must be defined as `int` or `unsigned`
  - Cannot access individual bits
- Defining bit fields
  - Follow `unsigned` or `int` member with a colon (`:`) and an integer constant representing the width of the field
  - Example:

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

© 2007 Pearson Education, Inc. All rights reserved.

10.10 Bit Fields

- **Unnamed bit field**
  - Field used as padding in the structure
  - Nothing may be stored in the bits

```
struct Example {
    unsigned a : 13;
    unsigned   : 3;
    unsigned b : 4;
}
```

  - Unnamed bit field with zero width aligns next bit field to a new storage unit boundary

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 10.3

Bit fields help conserve storage.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_16.c

(1 of 2)

```
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; /* new type name for struct bitCard */
14
15 void fillDeck( Card * const wDeck );    /* prototype */
16 void deal( const Card * const wDeck ); /* prototype */
17
18 int main( void )
19 {
20    Card deck[ 52 ]; /* create array of Cards */
21
22    fillDeck( deck );
23    deal( deck );
24
25    return 0; /* indicates successful termination */
26
27 } /* end main */
28
```

Bit fields determine how much memory each member of a structure can take up

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig10_16.c

(2 of 2)

```
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
41 } /* end function fillDeck */
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
50    /* loop through wDeck */
51    for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
52       printf( "Card:%3d  Suit:%2d  Color:%2d   ",
53          wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
54       printf( "Card:%3d  Suit:%2d  Color:%2d\n",
55          wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
56    } /* end for */
57
58 } /* end function deal */
```

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
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
```

© 2007 Pearson Education, Inc. All rights reserved.

Portability Tip 10.8

Bit-field manipulations are machine dependent. For example, some computers allow bit fields to cross word boundaries, whereas others do not.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.14

Attempting to access individual bits of a bit field as if they were elements of an array is a syntax error. Bit fields are not "arrays of bits."

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.15

Attempting to take the address of a bit field (the & operator may not be used with bit fields because they do not have addresses).

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 10.4

Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine language operations to access only portions of an addressable storage unit. This is one of many examples of the kinds of space–time trade-offs that occur in computer science.

© 2007 Pearson Education, Inc. All rights reserved.

10.11 Enumeration Constants

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at 0 and are incremented by 1
    - Values can be set explicitly with =
    - Need unique constant names
  - Example:

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

    - Creates a new type enum Months in which the identifiers are set to the integers 1 to 12
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)

© 2007 Pearson Education, Inc. All rights reserved.

```
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 {
7     JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
8
9  int main( void )
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
```

Enumeration sets the value of constant **JAN** to 1 and the following constants to 2, 3, 4…

Outline

fig10_18.c

(1 of 2)

© 2007 Pearson Education, Inc. All rights reserved.

```
18    /* loop through months */
19    for ( month = JAN; month <= DEC; month++ ) {
20       printf( "%2d%11s\n", month, monthName[ month ] );
21    } /* end for */
22
23    return 0; /* indicates successful termination */
24 } /* end main */
```

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

Like symbolic constants, enumeration constants are replaced by their values at compile time

Outline

fig10_18.c

(2 of 2)

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 10.16

Assigning a value to an enumeration constant after it has been defined is a syntax error.

© 2007 Pearson Education, Inc. All rights reserved.

Good Programming Practice 10.5

Use only uppercase letters in the names of enumeration constants. This makes these constants stand out in a program and reminds you that enumeration constants are not variables.

© 2007 Pearson Education, Inc. All rights reserved.