



# **C Structures, Unions, Bit Manipulations and Enumerations**

C How to Program, 6/e



## OBJECTIVES

In this chapter, you'll 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.



- 10.1** Introduction
- 10.2** Structure Definitions
- 10.3** Initializing Structures
- 10.4** Accessing Structure Members
- 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
- 10.10** Enumeration Constants



# 10.1 Introduction

- ▶ **Structures**—sometimes referred to as **aggregates**—are collections of related variables under one name.
- ▶ Structures may contain variables of many different data types—in contrast to arrays that contain only elements of the same data type.
- ▶ Structures are commonly used to define records to be stored in files (see Chapter 11, C File Processing).
- ▶ Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees (see Chapter 12, C Data Structures).



## 10.2 Structure Definitions

- ▶ Structures are **derived data types**—they are constructed using objects of other types.
- ▶ Consider the following structure definition:
  - **struct** card {  
    **char** \*face;  
    **char** \*suit;  
};
- ▶ Keyword **struct** introduces the structure definition.
- ▶ The identifier **card** is the **structure tag**, which names the structure definition and is used with the keyword **struct** to declare variables of the **structure type**.



## 10.2 Structure Definitions (Cont.)

- ▶ In this example, the structure type is `struct card`.
- ▶ Variables declared within the braces of the structure definition are the structure's **members**.
- ▶ Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict (we'll soon see why).
- ▶ Each structure definition must end with a semicolon.



## Common Programming Error 10.1

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



## 10.2 Structure Definitions (Cont.)

- ▶ The definition of `struct card` contains members `face` and `suit` of type `char *`.
- ▶ Structure members can be variables of the primitive data types (e.g., `int`, `float`, etc.), or aggregates, such as arrays and other structures.
- ▶ Structure members can be of many types.



## 10.2 Structure Definitions (Cont.)

- ▶ For example, the following `struct` contains character array members for an employee's first and last names, an `int` member for the employee's age, a `char` member that would contain 'M' or 'F' for the employee's gender and a `double` member for the employee's hourly salary:

- ```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourlySalary;  
};
```



## 10.2 Structure Definitions (Cont.)

- ▶ A structure cannot contain an instance of itself.
- ▶ For example, a variable of type `struct employee` cannot be declared in the definition for `struct employee`.
- ▶ A pointer to `struct employee`, however, may be included.
- ▶ For example,

```
• struct employee2 {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourlySalary;  
    struct employee2 person; /* ERROR */  
    struct employee2 *ePtr; /* pointer */  
};
```

- ▶ `struct employee2` contains an instance of itself (`person`), which is an error.



## 10.2 Structure Definitions (Cont.)

- ▶ Because `ePtr` is a pointer (to type `struct employee2`), it is permitted in the definition.
- ▶ A structure containing a member that is a pointer to the same structure type is referred to as a **self-referential structure**.
- ▶ Self-referential structures are used in Chapter 12 to build linked data structures.



## 10.2 Structure Definitions (Cont.)

- ▶ Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that is used to define variables.
- ▶ Structure variables are defined like variables of other types.
- ▶ The definition
  - **struct** card aCard, deck[ 52 ], \*cardPtr;  
declares aCard to be a variable of type **struct card**, declares **deck** to be an array with 52 elements of type **struct card** and declares **cardPtr** to be a pointer to **struct card**.



## 10.2 Structure Definitions (Cont.)

- ▶ Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.
- ▶ For example, the preceding definition could have been incorporated into the `struct card` structure definition as follows:

- ```
struct card {  
    char *face;  
    char *suit;  
} aCard, deck[ 52 ], *cardPtr;
```



## 10.2 Structure Definitions (Cont.)

- ▶ The structure tag name is optional.
- ▶ If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition—not in a separate declaration.



## 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.*



## Good Programming Practice 10.2

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



## 10.2 Structure Definitions (Cont.)

- ▶ The only valid operations that may be performed on structures are:
  - assigning structure variables to structure variables of the same type,
  - taking the address (&) of a structure variable,
  - accessing the members of a structure variable (see Section 10.4) and
  - using the `sizeof` operator to determine the size of a structure variable.



## Common Programming Error 10.2

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



## 10.2 Structure Definitions (Cont.)

- ▶ Structures may not be compared using operators `==` and `!=`, because structure members are not necessarily stored in consecutive bytes of memory.
- ▶ Sometimes there are “holes” in a structure, because computers may store specific data types only on certain memory boundaries such as half word, word or double word boundaries.
- ▶ A word is a standard memory unit used to store data in a computer—usually 2 bytes or 4 bytes.



## 10.2 Structure Definitions (Cont.)

- ▶ Consider the following structure definition, in which `sample1` and `sample2` of type `struct example` are declared:
  - ```
struct example {  
    char c;  
    int i;  
} sample1, sample2;
```
- ▶ A computer with 2-byte words may require that each member of `struct example` be aligned on a word boundary, i.e., at the beginning of a word (this is machine dependent).



## 10.2 Structure Definitions (Cont.)

- ▶ Figure 10.1 shows a sample storage alignment for a variable of type `struct example` that has been assigned the character 'a' and the integer 97 (the bit representations of the values are shown).
- ▶ If the members are stored beginning at word boundaries, there is a 1-byte hole (byte 1 in the figure) in the storage for variables of type `struct example`.
- ▶ The value in the 1-byte hole is undefined.
- ▶ Even if the member values of `sample1` and `sample2` are in fact equal, the structures are not necessarily equal, because the undefined 1-byte holes are not likely to contain identical values.



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



### **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.*



## 10.3 Initializing Structures

- ▶ Structures can be initialized using initializer lists as with arrays.
- ▶ To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
- ▶ For example, the declaration
  - `struct card aCard = { "Three", "Hearts" };`  
creates variable `aCard` to be of type `struct card` (as defined in Section 10.2) and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`.



## 10.3 Initializing Structures (Cont.)

- ▶ If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).
- ▶ Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they are not explicitly initialized in the external definition.
- ▶ Structure variables may also be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.



## 10.4 Accessing Structure Members

- ▶ Two operators are used to access members of structures: the **structure member operator (.)**—also called the dot operator—and the **structure pointer operator (->)**—also called the **arrow operator**.
- ▶ The structure member operator accesses a structure member via the structure variable name.
- ▶ For example, to print member `suit` of structure variable `aCard` defined in Section 10.3, use the statement
  - `printf( "%s", aCard.suit ); /* displays Hearts */`



## 10.4 Accessing Structure Members (Cont.)

- ▶ The structure pointer operator—consisting of a minus (–) sign and a greater than (>) sign with no intervening spaces—accesses a structure member via a **pointer to the structure**.
- ▶ Assume that the pointer `cardPtr` has been declared to point to `struct card` and that the address of structure `aCard` has been assigned to `cardPtr`.
- ▶ To print member `suit` of structure `aCard` with pointer `cardPtr`, use the statement
  - `printf( "%s", cardPtr->suit ); /* displays Hearts */`



## 10.4 Accessing Structure Members (Cont.)

- ▶ The expression `cardPtr->suit` is equivalent to `(*cardPtr).suit`, which dereferences the pointer and accesses the member `suit` using the structure member operator.
- ▶ The parentheses are needed here because the structure member operator `(.)` has a higher precedence than the pointer dereferencing operator `(*)`.
- ▶ The structure pointer operator and structure member operator, along with parentheses (for calling functions) and brackets `[]` used for array subscripting, have the highest operator precedence and associate from left to right.



### 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.*



### Common Programming Error 10.3

*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.*



## Common Programming Error 10.4

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



## Common Programming Error 10.5

*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.*



## 10.4 Accessing Structure Members (Cont.)

- ▶ The program of Fig. 10.2 demonstrates the use of the structure member and structure pointer operators.
- ▶ Using the structure member operator, the members of structure `aCard` are assigned the values "Ace" and "Spades", respectively (lines 18 and 19).
- ▶ Pointer `cardPtr` is assigned the address of structure `aCard` (line 21).
- ▶ Function `printf` prints the members of structure variable `aCard` using the structure member operator with variable name `aCard`, the structure pointer operator with pointer `cardPtr` and the structure member operator with dereferenced pointer `cardPtr` (lines 23 through 25).



```
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";
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,
```

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part I of 2.)



```
24     cardPtr->face, " of ", cardPtr->suit,  
25     ( *cardPtr ).face, " of ", ( *cardPtr ).suit );  
26     return 0; /* indicates successful termination */  
27 } /* end main */
```

Ace of Spades  
Ace of Spades  
Ace of Spades

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part 2 of 2.)



## 10.5 Using Structures with Functions

- ▶ Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.
- ▶ When structures or individual structure members are passed to a function, they are passed by value.
- ▶ Therefore, the members of a caller's structure cannot be modified by the called function.
- ▶ To pass a structure by reference, pass the address of the structure variable.



## 10.5 Using Structures with Functions (Cont.)

- ▶ Arrays of structures—like all other arrays—are automatically passed by reference.
- ▶ To pass an array by value, create a structure with the array as a member.
- ▶ Structures are passed by value, so the array is passed by value.



## Common Programming Error 10.6

*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.*



### **Performance Tip 10.1**

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



## 10.6 typedef

- ▶ The keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- ▶ Names for structure types are often defined with `typedef` to create shorter type names.
- ▶ For example, the statement
  - `typedef struct card Card;`  
defines the new type name `Card` as a synonym for type `struct card`.
- ▶ C programmers often use `typedef` to define a structure type, so a structure tag is not required.



## 10.6 typedef (Cont.)

- ▶ For example, the following definition

- ```
typedef struct {  
    char *face;  
    char *suit;  
} Card;
```

creates the structure type `Card` without the need for a separate `typedef` statement.



## Good Programming Practice 10.4

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



## 10.6 typedef (Cont.)

- ▶ `Card` can now be used to declare variables of type `struct card`.
- ▶ The declaration
  - `Card deck[ 52 ];`  
declares an array of 52 `Card` structures (i.e., variables of type `struct card`).
- ▶ Creating a new name with `typedef` does not create a new type; `typedef` simply creates a new type name, which may be used as an alias for an existing type name.



## 10.6 typedef (Cont.)

- ▶ A meaningful name helps make the program self-documenting.
- ▶ For example, when we read the previous declaration, we know “**deck** is an array of 52 **Cards**.”
- ▶ Often, **typedef** is used to create synonyms for the basic data types.
- ▶ For example, a program requiring 4-byte integers may use type **int** on one system and type **long** on another.
- ▶ Programs designed for portability often use **typedef** to create an alias for 4-byte integers, such as **Integer**.
- ▶ The alias **Integer** can be changed once in the program to make the program work on both systems.



## Portability Tip 10.2

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



## 10.7 Example: High-Performance Card Shuffling and Dealing Simulation

- ▶ The program in Fig. 10.3 is based on the card shuffling and dealing simulation discussed in Chapter 7.
- ▶ The program represents the deck of cards as an array of structures.
- ▶ The program uses high-performance shuffling and dealing algorithms.
- ▶ The output for the high-performance card shuffling and dealing program is shown in Fig. 10.4.



```
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 {
```

**Fig. 10.3** | High-performance card shuffling and dealing simulation. (Part I of 4.)



```
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
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 return 0; /* indicates successful termination */
39 } /* end main */
40
41 /* place strings into Card structures */
42 void fillDeck( Card * const wDeck, const char * wFace[],
43     const char * wSuit[] )
44 {
45     int i; /* counter */
46
```

**Fig. 10.3** | High-performance card shuffling and dealing simulation. (Part 2 of 4.)



```
47     /* loop through wDeck */
48     for ( i = 0; i <= 51; i++ ) {
49         wDeck[ i ].face = wFace[ i % 13 ];
50         wDeck[ i ].suit = wSuit[ i / 13 ];
51     } /* end for */
52 } /* end function fillDeck */
53
54 /* shuffle cards */
55 void shuffle( Card * const wDeck )
56 {
57     int i; /* counter */
58     int j; /* variable to hold random value between 0 - 51 */
59     Card temp; /* define temporary structure for swapping Cards */
60
61     /* loop through wDeck randomly swapping Cards */
62     for ( i = 0; i <= 51; i++ ) {
63         j = rand() % 52;
64         temp = wDeck[ i ];
65         wDeck[ i ] = wDeck[ j ];
66         wDeck[ j ] = temp;
67     } /* end for */
68 } /* end function shuffle */
69
```

**Fig. 10.3** | High-performance card shuffling and dealing simulation. (Part 3 of 4.)



---

```
70  /* deal cards */
71  void deal( const Card * const wDeck )
72  {
73      int i; /* counter */
74
75      /* loop through wDeck */
76      for ( i = 0; i <= 51; i++ ) {
77          printf( "%5s of %-8s%s", wDeck[ i ].face, wDeck[ i ].suit,
78                  ( i + 1 ) % 4 ? " " : "\n" );
79      } /* end for */
80  } /* end function deal */
```

---

**Fig. 10.3** | High-performance card shuffling and dealing simulation. (Part 4 of 4.)



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

**Fig. 10.4** | Output for the high-performance card shuffling and dealing simulation.



## 10.7 Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- ▶ In the program, function `fillDeck` (lines 42–52) initializes the `Card` array in order with Ace through King of each suit.
- ▶ The `Card` array is passed (in line 36) to function `shuffle` (lines 55–68), where the high-performance shuffling algorithm is implemented.
- ▶ Function `shuffle` takes an array of 52 `Card` structures as an argument.
- ▶ The function loops through the 52 cards (array subscripts 0 to 51) using a `for` statement in lines 62–67.



## 10.7 Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- ▶ For each card, a number between 0 and 51 is picked randomly.
- ▶ Next, the current **Card** structure and the randomly selected **Card** structure are swapped in the array (lines 64 through 66).
- ▶ A total of 52 swaps are made in a single pass of the entire array, and the array of **Card** structures is shuffled!
- ▶ This algorithm cannot suffer from indefinite postponement like the shuffling algorithm presented in Chapter 7.
- ▶ Since the **Card** structures were swapped in place in the array, the high-performance dealing algorithm implemented in function **deal** (lines 71–80) requires only one pass of the array to deal the shuffled cards.



## Common Programming Error 10.7

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



## 10.8 Unions

- ▶ A **union** is a derived data type—like a structure—with members that share the same storage space.
- ▶ For different situations in a program, some variables may not be relevant, but other variables are—so a union shares the space instead of wasting storage on variables that are not being used.
- ▶ The members of a union can be of any data type.



## 10.8 Unions (Cont.)

- ▶ The number of bytes used to store a union must be at least enough to hold the largest member.
- ▶ In most cases, unions contain two or more data types.
- ▶ Only one member, and thus one data type, can be referenced at a time.
- ▶ It is your responsibility to ensure that the data in a union is referenced with the proper data type.



## Common Programming Error 10.8

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



### **Portability Tip 10.3**

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



## 10.8 Unions (Cont.)

- ▶ A union is declared with keyword `union` in the same format as a structure.
- ▶ The `union` definition
  - `union number {  
    int x;  
    double y;  
};`
- ▶ indicates that `number` is a `union` type with members `int x` and `double y`.
- ▶ The union definition is normally placed in a header and included in all source files that use the union type.



## 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.*



## 10.8 Unions (Cont.)

- ▶ The operations that can be performed on a union are the following:
  - assigning a union to another union of the same type,
  - taking the address (&) of a union variable,
  - and accessing union members using the structure member operator and the structure pointer operator.
- ▶ Unions may not be compared using operators == and != for the same reasons that structures cannot be compared.



## 10.8 Unions (Cont.)

- ▶ In a declaration, a union may be initialized with a value of the same type as the first union member.
- ▶ For example, with the preceding union, the declaration
  - `union number value = { 10 };`is a valid initialization of union variable `value` because the union is initialized with an `int`, but the following declaration would truncate the floating-point part of the initializer value and normally would produce a warning from the compiler:



union 1.03



## Common Programming Error 10.9

*Comparing unions is a syntax error.*



### **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.*



### **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.*



## **Performance Tip 10.2**

*Unions conserve storage.*



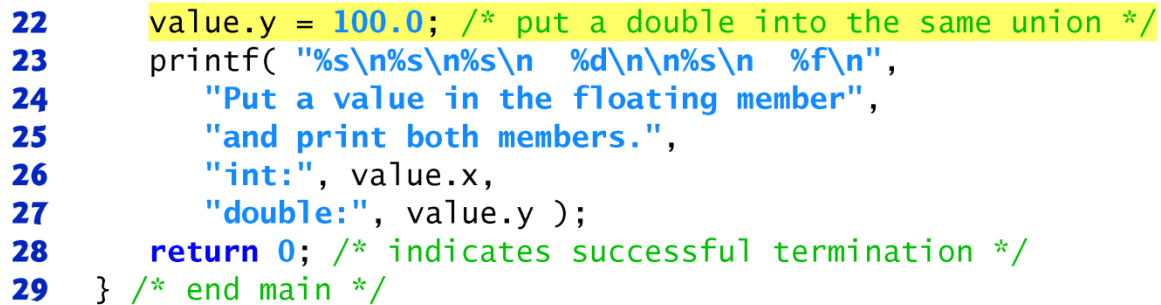
## 10.8 Unions (Cont.)

- ▶ The program in Fig. 10.5 uses the variable `value` (line 13) of type `union number` to display the value stored in the union as both an `int` and a `double`.
- ▶ The program output is implementation dependent.
- ▶ The program output shows that the internal representation of a `double` value can be quite different from the representation of `int`.



```
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
```

**Fig. 10.5** | Displaying the value of a union in both member data types. (Part I of 2.)

[illegible]

```
int:
    0

double:
    100.000000
```

©1992–2010 by Pearson Education, Inc.  
All Rights Reserved.



## 10.9 Bitwise Operators

- ▶ Computers represent all data internally as sequences of bits.
- ▶ Each bit can assume the value 0 or the value 1.
- ▶ On most systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type `char`.
- ▶ Other data types are stored in larger numbers of bytes.
- ▶ The bitwise operators are used to manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both `signed` and `unsigned`).
- ▶ Unsigned integers are normally used with the bitwise operators.



## Portability Tip 10.6

*Bitwise data manipulations are machine dependent.*



## 10.9 Bitwise Operators (Cont.)

- ▶ The bitwise operator discussions in this section show the binary representations of the integer operands.
- ▶ For a detailed explanation of the binary (also called base-2) number system see Appendix C.
- ▶ Also, the programs in Sections 10.9–10.10 were tested using Microsoft Visual C++.
- ▶ Because of the machine-dependent nature of bitwise manipulations, these programs may not work on your system.
- ▶ The bitwise operators are **bitwise AND (&)**, **bitwise inclusive OR (|)**, **bitwise exclusive OR (^)**, **left shift (<<)**, **right shift (>>)** and **complement (~)**.



## 10.9 Bitwise Operators (Cont.)

- ▶ The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit.
- ▶ The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands is 1.
- ▶ The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1.
- ▶ The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1.



## 10.9 Bitwise Operators (Cont.)

- ▶ The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- ▶ The right-shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand.
- ▶ The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits to 0 in the result.
- ▶ Detailed discussions of each bitwise operator appear in the following examples.
- ▶ The bitwise operators are summarized in Fig. 10.6.



| 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.



## 10.9 Bitwise Operators (Cont.)

- ▶ When using the bitwise operators, it is useful to print values in their binary representation to illustrate the precise effects of these operators.
- ▶ The program of Fig. 10.7 prints an `unsigned` integer in its binary representation in groups of eight bits each.
- ▶ For the examples in this section, we assume that `unsigned` integers are stored in 4 bytes (32 bits) of memory.



```
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     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     /* define displayMask and left shift 31 bits */
24     unsigned displayMask = 1 << 31;
25
```

**Fig. 10.7** | Displaying an unsigned integer in bits. (Part I of 2.)



```
26     printf( "%10u = ", value );
27
28     /* loop through bits */
29     for ( c = 1; c <= 32; c++ ) {
30         putchar( value & displayMask ? '1' : '0' );
31         value <<= 1; /* shift value left by 1 */
32
33         if ( c % 8 == 0 ) { /* output space after 8 bits */
34             putchar( ' ' );
35         } /* end if */
36     } /* end for */
37
38     putchar( '\n' );
39 } /* end function displayBits */
```

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

**Fig. 10.7** | Displaying an unsigned integer in bits. (Part 2 of 2.)



## 10.9 Bitwise Operators (Cont.)

- ▶ Function `displayBits` (lines 19–39) uses the bitwise AND operator to combine variable `value` with variable `displayMask` (line 32).
- ▶ Often, the bitwise AND operator is used with an operand called a **mask**—an integer value with specific bits set to 1.
- ▶ Masks are used to hide some bits in a value while selecting other bits.
- ▶ In function `displayBits`, mask variable `displayMask` is assigned the value
  - `1 << 31` (10000000 00000000 00000000 00000000)



## 10.9 Bitwise Operators (Cont.)

- ▶ The left-shift operator shifts the value `1` from the low order (rightmost) bit to the high order (leftmost) bit in `displayMask` and fills in `0` bits from the right.
- ▶ Line 32
  - `putchar( value & displayMask ? '1' : '0' );`  
determines whether a `1` or a `0` should be printed for the current leftmost bit of variable `value`.
- ▶ When `value` and `displayMask` are combined using `&`, all the bits except the high-order bit in variable `value` are “masked off” (hidden), because any bit “ANDed” with `0` yields `0`.



## 10.9 Bitwise Operators (Cont.)

- ▶ If the leftmost bit is 1, `value & displayMask` evaluates to a nonzero (true) value and 1 is printed—otherwise, 0 is printed.
- ▶ Variable `value` is then left shifted one bit by the expression `value <<= 1` (this is equivalent to `value = value << 1`).
- ▶ These steps are repeated for each bit in unsigned variable `value`.
- ▶ Figure 10.8 summarizes the results of combining two bits with the bitwise AND operator.



| 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 &.



## Common Programming Error 10.10

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



## 10.9 Bitwise Operators (Cont.)

- ▶ In line 24 of Fig. 10.7, we hard coded the integer 31 to indicate that the value 1 should be shifted to the leftmost bit in the variable `displayMask`.
- ▶ Similarly, in line 29, we hard coded the integer 32 to indicate that the loop should iterate 32 times—once for each bit in variable `value`.
- ▶ We assumed that `unsigned` integers are always stored in 32 bits (4 bytes) of memory.
- ▶ Many of today's popular computers use 32-bit word hardware architectures.



## 10.9 Bitwise Operators (Cont.)

- ▶ C programmers tend to work across many hardware architectures, and sometimes **unsigned** integers will be stored in smaller or larger numbers of bits.
- ▶ The program in Fig. 10.7 can be made more scalable and more portable by replacing the integer **31** in line 24 with the expression
  - **CHAR\_BIT \* sizeof( unsigned ) - 1**and by replacing the integer **32** in line 29 with the the expression
  - **CHAR\_BIT \* sizeof( unsigned )**



## 10.9 Bitwise Operators (Cont.)

- ▶ The symbolic constant `CHAR_BIT` (defined in `<limits.h>`) represents the number of bits in a byte (normally 8).
- ▶ On a computer that uses 32-bit words, the expression `sizeof( unsigned )` evaluates to 4, so the two preceding expressions evaluate to 31 and 32, respectively.
- ▶ On a computer that uses 16-bit words, the `sizeof` expression evaluates to 2 and the two preceding expressions evaluate to 15 and 16, respectively.



## 10.9 Bitwise Operators (Cont.)

- ▶ Figure 10.9 demonstrates the use of the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator.
- ▶ The program uses function `displayBits` (lines 53–74) to print the `unsigned` integer values.
- ▶ The output is shown in Fig. 10.10.



---

```
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 );
```

---

**Fig. 10.9** | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part I of 4.)



```
23
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
```

**Fig. 10.9** | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 4.)



---

```
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  return 0; /* indicates successful termination */
49  } /* end main */
50
```

---

**Fig. 10.9** | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 3 of 4.)



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

**Fig. 10.9** | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 4 of 4.)



```
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
```

**Fig. 10.10** | Output for the program of Fig. 10.9.



## 10.9 Bitwise Operators (Cont.)

- ▶ In Fig. 10.9, integer variable `number1` is assigned value 65535 (00000000 00000000 11111111 11111111) in line 16 and variable `mask` is assigned the value 1 (00000000 00000000 00000000 00000001) in line 17.
- ▶ When `number1` and `mask` are combined using the bitwise AND operator (&) in the expression `number1 & mask` (line 22), the result is 00000000 00000000 00000000 00000001.
- ▶ All the bits except the low-order bit in variable `number1` are “masked off” (hidden) by “ANDing” with variable `mask`.
- ▶ The bitwise inclusive OR operator is used to set specific bits to 1 in an operand.



## 10.9 Bitwise Operators (Cont.)

- ▶ In Fig. 10.9, variable `number1` is assigned 15 (00000000 00000000 00000000 00001111) in line 25, and variable `setBits` is assigned 241 (00000000 00000000 00000000 11110001) in line 26.
- ▶ When `number1` and `setBits` are combined using the bitwise OR operator in the expression `number1 | setBits` (line 31), the result is 255 (00000000 00000000 00000000 11111111).
- ▶ Figure 10.11 summarizes the results of combining two bits with the bitwise inclusive OR operator.



## Common Programming Error 10.11

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



| 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 Bitwise Operators (Cont.)

- ▶ The bitwise exclusive OR operator ( $\wedge$ ) sets each bit in the result to 1 if *exactly* one of the corresponding bits in its two operands is 1.
- ▶ In Fig. 10.9, variables `number1` and `number2` are assigned the values 139 (00000000 00000000 00000000 10001011) and 199 (00000000 00000000 00000000 11000111) in lines 34–35.
- ▶ When these variables are combined with the exclusive OR operator in the expression `number1 ^ number2` (line 40), the result is 00000000 00000000 00000000 01001100.
- ▶ Figure 10.12 summarizes the results of combining two bits with the bitwise exclusive OR operator.



| Bit 1 | Bit 2 | Bit 1 $\wedge$ 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  $\wedge$ .



## 10.9 Bitwise Operators (Cont.)

- ▶ The **bitwise complement operator** ( $\sim$ ) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1 in the result—otherwise referred to as “taking the **one’s complement** of the value.”
- ▶ In Fig. 10.9, variable `number1` is assigned the value 21845 (00000000 00000000 01010101 01010101) in line 43.
- ▶ When the expression `~number1` (line 47) is evaluated, the result is 00000000 00000000 10101010 10101010.



## 10.9 Bitwise Operators (Cont.)

- ▶ The program of Fig. 10.13 demonstrates the left-shift operator (<<) and the right-shift operator (>>).
- ▶ Function `displayBits` is used to print the unsigned integer values.



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

**Fig. 10.13** | Bitwise shift operators. (Part I of 3.)



```
26
27  /* display bits of an unsigned integer value */
28  void displayBits( unsigned value )
29  {
30      unsigned c; /* counter */
31
32      /* declare displayMask and left shift 31 bits */
33      unsigned displayMask = 1 << 31;
34
35      printf( "%7u = ", value );
36
37      /* loop through bits */
38      for ( c = 1; c <= 32; c++ ) {
39          putchar( value & displayMask ? '1' : '0' );
40          value <<= 1; /* shift value left by 1 */
41
42          if ( c % 8 == 0 ) { /* output a space after 8 bits */
43              putchar( ' ' );
44          } /* end if */
45      } /* end for */
46
47      putchar( '\n' );
48  } /* end function displayBits */
```

**Fig. 10.13** | Bitwise shift operators. (Part 2 of 3.)



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

**Fig. 10.13** | Bitwise shift operators. (Part 3 of 3.)



## 10.9 Bitwise Operators (Cont.)

- ▶ The left-shift operator ( $\ll$ ) shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- ▶ Bits vacated to the right are replaced with 0s; 1s shifted off the left are lost.
- ▶ In Fig. 10.13, variable `number1` is assigned the value 960 (00000000 00000000 00000011 11000000) in line 9.
- ▶ The result of left shifting variable `number1` 8 bits in the expression `number1 << 8` (line 16) is 49152 (00000000 00000000 11000000 00000000).



## 10.9 Bitwise Operators (Cont.)

- ▶ The right-shift operator ( $\gg$ ) shifts the bits of its left operand to the right by the number of bits specified in its right operand.
- ▶ Performing a right shift on an **unsigned** integer causes the vacated bits at the left to be replaced by 0s; 1s shifted off the right are lost.
- ▶ In Fig. 10.13, the result of right shifting **number1** in the expression **number1  $\gg$  8** (line 23) is 3 (00000000 00000000 00000000 00000011).



## Common Programming Error 10.12

*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.*



### 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.*



## 10.9 Bitwise Operators (Cont.)

- ▶ Each binary bitwise operator has a corresponding assignment operator.
- ▶ These **bitwise assignment operators** are shown in Fig. 10.14 and are used in a manner similar to the arithmetic assignment operators introduced in Chapter 3.



### Bitwise assignment operators

|                        |                                           |
|------------------------|-------------------------------------------|
| <code>&amp;=</code>    | Bitwise AND assignment operator.          |
| <code> =</code>        | Bitwise inclusive OR assignment operator. |
| <code>^=</code>        | Bitwise exclusive OR assignment operator. |
| <code>&lt;&lt;=</code> | Left-shift assignment operator.           |
| <code>&gt;&gt;=</code> | Right-shift assignment operator.          |

**Fig. 10.14** | The bitwise assignment operators.



## 10.9 Bitwise Operators (Cont.)

- ▶ Figure 10.15 shows the precedence and associativity of the various operators introduced to this point in the text.
- ▶ They are shown top to bottom in decreasing order of precedence.



| Operator                                 | Associativity | Type           |
|------------------------------------------|---------------|----------------|
| () [] . ->                               | left to right | highest        |
| + - ++ -- ! & * ~ sizeof ( <i>type</i> ) | 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 Bit Fields

- ▶ C enables you to specify the number of bits in which an `unsigned` or `int` member of a structure or union is stored.
- ▶ This is referred to as a `bit field`.
- ▶ Bit fields enable better memory utilization by storing data in the minimum number of bits required.
- ▶ Bit field members *must* be declared as `int` or `unsigned`.



### **Performance Tip 10.3**

*Bit fields help conserve storage.*



## 10.10 Bit Fields (Cont.)

- ▶ Consider the following structure definition:

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

which contains three `unsigned` bit fields—`face`, `suit` and `color`—used to represent a card from a deck of 52 cards.



## 10.10 Bit Fields (Cont.)

- ▶ A bit field is declared by following an `unsigned` or `int` `member name` with a colon (`:`) and an integer constant representing the `width` of the field (i.e., the number of bits in which the member is stored).
- ▶ The constant representing the width must be an integer between 0 and the total number of bits used to store an `int` on your system, inclusive.
- ▶ Our examples were tested on a computer with 4-byte (32-bit) integers.
- ▶ The preceding structure definition indicates that member `face` is stored in 4 bits, member `suit` is stored in 2 bits and member `color` is stored in 1 bit.



## 10.10 Bit Fields (Cont.)

- ▶ The number of bits is based on the desired range of values for each structure member.
- ▶ Member `face` stores values from 0 (Ace) through 12 (King)—4 bits can store values in the range 0–15.
- ▶ Member `suit` stores values from 0 through 3 (0 = Diamonds, 1 = Hearts, 2 = Clubs, 3 = Spades)—2 bits can store values in the range 0–3.
- ▶ Finally, member `color` stores either 0 (Red) or 1 (Black)—1 bit can store either 0 or 1.
- ▶ Figure 10.16 (output shown in Fig. 10.17) creates array `deck` containing 52 `struct bitCard` structures in line 20.



## 10.10 Bit Fields (Cont.)

- ▶ Function `fillDeck` (lines 28–38) inserts the 52 cards in the `deck` array and function `deal` (lines 42–54) prints the 52 cards.
- ▶ Notice that bit field members of structures are accessed exactly as any other structure member.
- ▶ Member `color` is included as a means of indicating the card color on a system that allows color displays. It is possible to specify an **unnamed bit field** to be used as **padding** in the structure.

## 10.10 Bit Fields (Cont.)

- ▶ For example, the structure definition

- ```
struct example {  
    unsigned a : 13;  
    unsigned   : 19;  
    unsigned b : 4;  
};
```

uses an unnamed 19-bit field as padding—nothing can be stored in those 19 bits.

- ▶ Member **b** (on our 4-byte-word computer) is stored in another storage unit.



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

**Fig. 10.16** | Bit fields to store a deck of cards. (Part 1 of 3.)



```
26
27  /* initialize Cards */
28  void fillDeck( Card * const wDeck )
29  {
30      int i; /* counter */
31
32      /* loop through wDeck */
33      for ( i = 0; i <= 51; i++ ) {
34          wDeck[ i ].face = i % 13;
35          wDeck[ i ].suit = i / 13;
36          wDeck[ i ].color = i / 26;
37      } /* end for */
38  } /* end function fillDeck */
39
```

**Fig. 10.16** | Bit fields to store a deck of cards. (Part 2 of 3.)



```
40  /* output cards in two column format; cards 0-25 subscripted with
41     k1 (column 1); cards 26-51 subscripted k2 (column 2) */
42  void deal( const Card * const wDeck )
43  {
44      int k1; /* subscripts 0-25 */
45      int k2; /* subscripts 26-51 */
46
47      /* loop through wDeck */
48      for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
49          printf( "Card:%3d Suit:%2d Color:%2d  ",
50                 wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
51          printf( "Card:%3d Suit:%2d Color:%2d\n",
52                 wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
53      } /* end for */
54  } /* end function deal */
```

**Fig. 10.16** | Bit fields to store a deck of cards. (Part 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 |

**Fig. 10.17** | Output of the program in Fig. 10.16.

## 10.10 Bit Fields (Cont.)

- ▶ An **unnamed bit field with a zero width** is used to align the next bit field on a new storage-unit boundary.
- ▶ For example, the structure definition
  - **struct example {**  
    **unsigned a : 13;**  
    **unsigned : 0;**  
    **unsigned b : 4;**  
};

uses an unnamed 0-bit field to skip the remaining bits (as many as there are) of the storage unit in which **a** is stored and to align **b** on the next storage-unit boundary.



### **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.*



### Common Programming Error 10.13

*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.”*



### Common Programming Error 10.14

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



### **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.*

## 10.10 Enumeration Constants

- ▶ C provides one final user-defined type called an **enumeration**.
- ▶ An enumeration, introduced by the keyword **enum**, is a set of integer **enumeration constants** represented by identifiers.
- ▶ Values in an **enum** start with 0, unless specified otherwise, and are incremented by 1.
- ▶ For example, the enumeration
  - **enum** months {  
    **JAN**, **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 0 to 11, respectively.

## 10.11 Enumeration Constants (Cont.)

- ▶ To number the months 1 to 12, use the following enumeration:
  - `enum months {  
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,  
    SEP, OCT, NOV, DEC };`
- ▶ Since the first value in the preceding enumeration is explicitly set to 1, the remaining values are incremented from 1, resulting in the values 1 through 12.
- ▶ The identifiers in an enumeration must be unique.
- ▶ The value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier.



## 10.12 Enumeration Constants

- ▶ Multiple members of an enumeration can have the same constant value.
- ▶ In the program of Fig. 10.18, the enumeration variable `month` is used in a `for` statement to print the months of the year from the array `monthName`.
- ▶ We've made `monthName[0]` the empty string `""`.
- ▶ Some programmers might prefer to set `monthName[0]` to a value such as `***ERROR***` to indicate that a logic error occurred.



## Common Programming Error 10.15

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



## Good Programming Practice 10.5

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



```
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
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 */
```

**Fig. 10.18** | Using an enumeration. (Part I of 2.)



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

**Fig. 10.18** | Using an enumeration. (Part 2 of 2.)