Chapter 10
C Structures, Unions, Bit Manipulation and Enumerations

C How to Program, 8/e, GE

Objectives

In this chapter, you'll:

- Create and use `struct`s, `union`s and `enum`s.
- Understand self-referential `struct`s.
- Learn about the operations that can be performed on `struct` instances.
- Initialize `struct` members.
- Access `struct` members.
- Pass `struct` instances to functions by value and by reference.
- Use `typedef`s to create aliases for existing type names.
- Learn about the operations that can be performed on `union`s.
- Initialize `union`s.
- Manipulate integer data with the bitwise operators.
- Create bit fields for storing data compactly.
- Use `enum` constants.
- Consider the security issues of working with `struct`s, bit manipulation and `enum`s.

10.1 Introduction

10.2 Structure Definitions

- 10.2.1 Self-Referential Structures
- 10.2.2 Defining Variables of Structure Types
- 10.2.3 Structure Tag Names
- 10.2.4 Operations That Can Be Performed on Structures

10.3 Initializing Structures

10.4 Accessing Structure Members with `.` and `->`

10.5 Using Structures with Functions

10.6 `typedef`

10.7 Example: High-Performance Card Shuffling and Dealing Simulation

10.8 Unions

- 10.8.1 Union Declarations
- 10.8.2 Operations That Can Be Performed on Unions
- 10.8.3 Initializing Unions in Declarations
- 10.8.4 Demonstrating Unions

10.9 Bitwise Operators
- 10.9.1 Displaying an Unsigned Integer in Bits
- 10.9.2 Making Function `displayBits` More Generic and Portable
- 10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators
- 10.9.4 Using the Bitwise Left- and Right-Shift Operators
- 10.9.5 Bitwise Assignment Operators

10.10 Bit Fields
- 10.10.1 Defining Bit Fields
- 10.10.2 Using Bit Fields to Represent a Card's Face, Suit and Color
- 10.10.3 Unnamed Bit Fields

10.11 Enumeration Constants

10.12 Anonymous Structures and Unions

10.13 Secure C Programming

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, which 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).

10.1 Introduction (Cont.)

- We'll also discuss:
  - `typedefs`—for creating *aliases* for previously defined data types
  - `unions`—derived data types like structures, but with members that *share* the *same* storage space
  - bitwise operators—for manipulating the bits of integral operands
  - bit fields—`unsigned int` or `int` members of structures or unions for which you specify the number of bits in which the members are stored, helping you pack information tightly
  - enumerations—sets of integer constants represented by identifiers.

10.2 Structure Definitions

- Structures are derived data types—they're 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 `struct` to declare variables of the structure type—e.g., `struct card`.

10.2 Structure Definitions (Cont.)

- 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`, each 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 `unsigned 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];
   unsigned int age;
   char gender;
   double hourlySalary;
};
```

10.2.1 Self-Referential Structures

- *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];
   unsigned 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's permitted in the definition.
- A structure containing a member that's 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.

10.2.2 Defining Variables of Structure Types

- Structure definitions do *not* reserve any space in memory; rather, each definition creates a new data type that's 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` definition as follows:

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

10.2.3 Structure Tag Names

- 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.

Common Programming Error 10.2

A structure cannot contain an instance of itself.

Good Programming Practice 10.1

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

10.2.4 Operations That Can Be Performed on Structures

- 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 10.3

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

10.2.4 Operations That Can Be Performed on Structures (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's 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.

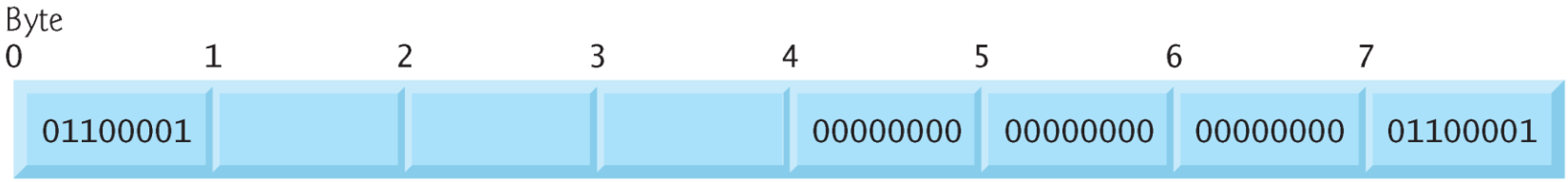

Fig. 10.1 | Possible storage alignment for a variable of type `struct` example showing an undefined area in memory.

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're 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 10.2

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 10.4

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.*

Common Programming Error 10.5

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

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. To prevent this problem use the arrow (->) operator instead.

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
- Pointer `cardPtr` is assigned the address of structure `aCard`
- 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`

```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

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

int main(void)
{
   struct card aCard; // define one struct card variable

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

```

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 1 of 2.)

```
22      printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
23         cardPtr->face, " of ", cardPtr->suit,
24         (*cardPtr).face, " of ", (*cardPtr).suit);
25   }
```

```
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're 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 10.7

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.

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 10.3

Capitalize the first letter of `typedef` *names to emphasize that they're 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 four-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 four-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.*

Good Programming Practice 10.4

Using `typedef`*s can help make a program more readable and maintainable.*

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 and uses high-performance shuffling and dealing algorithms.
- The program output is shown in Fig. 10.4.

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

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 1 of 4.)

```
24  int main(void)
25  {
26     Card deck[CARDS]; // define array of Cards
27
28     // initialize array of pointers
29     const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
30        "Six", "Seven", "Eight", "Nine", "Ten",
31        "Jack", "Queen", "King"};
32
33     // initialize array of pointers
34     const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
35
36     srand(time(NULL)); // randomize
37
38     fillDeck(deck, face, suit); // load the deck with Cards
39     shuffle(deck); // put Cards in random order
40     deal(deck); // deal all 52 Cards
41  }
42
```

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 2 of 4.)

```
43  // place strings into Card structures
44  void fillDeck(Card * const wDeck, const char * wFace[],
45     const char * wSuit[])
46  {
47     // loop through wDeck
48     for (size_t i = 0; i < CARDS; ++i) {
49        wDeck[i].face = wFace[i % FACES];
50        wDeck[i].suit = wSuit[i / FACES];
51     }
52  }
53
54  // shuffle cards
55  void shuffle(Card * const wDeck)
56  {
57     // loop through wDeck randomly swapping Cards
58     for (size_t i = 0; i < CARDS; ++i) {
59        size_t j = rand() % CARDS;
60        Card temp = wDeck[i];
61        wDeck[i] = wDeck[j];
62        wDeck[j] = temp;
63     }
64  }
65
```

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 3 of 4.)

```
66  // deal cards
67  void deal(const Card * const wDeck)
68  {
69     // loop through wDeck
70     for (size_t i = 0; i < CARDS; ++i) {
71        printf("%5s of %-8s%s", wDeck[i].face  , wDeck[i].suit  ,
72           (i + 1) % 4 ? "  " : "\n");
73     }
74  }
```

Fig. 10.3 | Card shuffling and dealing program using structures. (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` initializes the `Card` array in order with "Ace" through "King" of each suit.
- The `Card` array is passed to function `shuffle`, where the high-performance shuffling algorithm is implemented.
- Function `shuffle` takes an array of 52 `Card`s as an argument.
- The function loops through the 52 `Card`s

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` and the randomly selected `Card` are swapped in the array
- A total of 52 swaps are made in a single pass of the entire array, and the array of `Cards` is shuffled!
- This algorithm *cannot* suffer from *indefinite postponement* like the shuffling algorithm presented in Chapter 7.
- Because the `Cards` were swapped in place in the array, the high-performance dealing algorithm implemented in function `deal` requires only *one* pass of the array to deal the shuffled `Cards` .

Common Programming Error 10.8

Forgetting to include the array index 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's your responsibility to ensure that the data in a union is referenced with the proper data type.

Common Programming Error 10.9

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.

10.8.1 Union Declarations

- A union definition has the same format as a structure definition.
- 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.*

10.8.2 Operations That Can Be Performed on Unions

- The operations that can be performed on a union are:
  - 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.

10.8.3 Initializing Unions in Declarations

- In a declaration, *a union may be initialized with a value of the same type as the first union member*.
- For example, with the union in Section 10.8.1, the statement
  - `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 number value = {1.43};`

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 among 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.

10.8.4 Demonstrating Unions

- The program in Fig. 10.5 uses the variable `value` 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`.

```c
// Fig. 10.5: fig10_05.c
// Displaying the value of a union in both member data types
#include <stdio.h>

// number union definition
union number {
   int x;
   double y;
};

int main(void)
{
   union number value; // define union variable

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

Fig. 10.5 | Displaying the value of a union in both member data types. (Part 1 of 2.)

```
21
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 100.0 in the floating member",
25         "and print both members.",
26         "int:", value.x,
27         "double:", value.y);
28   }
```

```
Put 100 in the integer member
and print both members.
int:
  100

double:
  -92559592117433136000000000000000000000000000000000000000000000.000000


Put 100.0 in the floating member
and print both members.
int:
  0

double:
  100.000000
```

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

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 typical 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 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.
- Because of the machine-dependent nature of bitwise manipulations, these programs may not work correctly on your system.
- The bitwise operators are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^; also known as bitwise XOR), 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 examples that follow.
- The bitwise operators are summarized in Fig. 10.6.

Operator		Description
`&`	bitwise AND	Compares its two operands bit by bit. The bits in the result are set to `1` if the corresponding bits in the two operands are *both* `1`.
`|`	bitwise inclusive OR	Compares its two operands bit by bit. 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 (also known as bitwise XOR)	Compares its two operands bit by bit. The bits in the result are set to `1` if the corresponding bits in the two operands are different.
`<<`	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 when the left operand is negative.
`~`	complement	All `0` bits are set to `1` and all `1` bits are set to `0`.

Fig. 10.6 | Bitwise operators.

10.9.1 Displaying an Unsigned Integer in Bits

- When using the bitwise operators, it's useful to display values in binary to show the precise effects of these operators.
- The program of Fig. 10.7 prints an `unsigned int` in its binary representation in groups of eight bits each for readability.
- For the examples in this section, we assume an implementation where `unsigned ints` are stored in 4 bytes (32 bits) of memory.

```c
// Fig. 10.7: fig10_07.c
// Displaying an unsigned int in bits
#include <stdio.h>

void displayBits(unsigned int value); // prototype

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

   printf("%s", "Enter a nonnegative int: ");
   scanf("%u", &x);

   displayBits(x);
}

```

Fig. 10.7 | Displaying an `unsigned int` in bits. (Part 1 of 2.)

```
17  // display bits of an unsigned int value
18  void displayBits(unsigned int value)
19  {
20     // define displayMask and left shift 31 bits
21     unsigned int displayMask = 1 << 31;
22
23     printf("%10u = ", value);
24
25     // loop through bits
26     for (unsigned int c = 1; c <= 32; ++c) {
27        putchar(value & displayMask ? '1' : '0');
28        value <<= 1; // shift value left by 1
29
30        if (c % 8 == 0) { // output space after 8 bits
31           putchar(' ');
32        }
33     }
34
35     putchar('\n');
36  }
```

```
Enter a nonnegative int: 65000
     65000 = 00000000 00000000 11111101 11101000
```

Fig. 10.7 | Displaying an `unsigned int` in bits. (Part 2 of 2.)

10.9.1 Displaying an Unsigned Integer in Bits (Cont.)

- Function `displayBits` uses the bitwise AND operator to combine variable `value` with variable `displayMask`
- 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`)

10.9.1 Displaying an Unsigned Integer in Bits (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.
- `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`.

10.9.1 Displaying an Unsigned Integer in Bits (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.

Common Programming Error 10.10

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

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

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

Portability Tip 10.7

Figure 10.7 can be made more generic and portable by replacing the integers `31` *(line 21) and* `32` *(line 26) with expressions that calculate these integers, based on the size of an* `unsigned int` *for the platform on which the program executes. The symbolic constant* `CHAR_BIT` *(defined in* `<limits.h>`*) represents the number of bits in a byte (normally 8). Recall* `sizeof` *determines the number of bytes used to store an object or type. The expression* `sizeof(unsigned int)` *evaluates to* `4` *for 32-bit* `unsigned int`*s and* `8` *for 64-bit* `unsigned int`*s. You can replace* `31` *with* `CHAR_BIT * sizeof(unsigned int) - 1` *and replace* `32` *with* `CHAR_BIT * sizeof(unsigned int)`*. For 32-bit* `unsigned int`*s, these expressions evaluate to* `31` *and* `32`*, respectively. For 64-bit* `unsigned int`*s, they evaluate to* `63` *and* `64`*.*

10.9.2 Making Function displayBits More Scalable and Portable

- In 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, 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 ints` are always stored in 32 bits (4 bytes) of memory.
- Many of today's popular computers use 32-bit or 64-bit word hardware architectures.

10.9.2 Making Function displayBits More Scalable and Portable (Cont.)

- As a C programmer, you'll tend to work across many hardware architectures, and sometimes `unsigned ints` will be stored in smaller or larger numbers of bits.
- We can make the program in Fig. 10.7 more scalable and more portable by replacing the integer 31 with the expression
  - `CHAR_BIT * sizeof(unsigned int) - 1`

  and by replacing the integer 32 with the expression
  - `CHAR_BIT * sizeof(unsigned int)`

10.9.2 Making Function displayBits More Scalable and Portable (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 int)` 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.

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators

- 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` to print the `unsigned int` values.
- The output is shown in Fig. 10.10.

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

void displayBits(unsigned int value); // prototype

int main(void)
{
   // demonstrate bitwise AND (&)
   unsigned int number1 = 65535;
   unsigned int mask = 1;
   puts("The result of combining the following");
   displayBits(number1);
   displayBits(mask);
   puts("using the bitwise AND operator & is");
   displayBits(number1 & mask);

```

Fig. 10.9 | Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 4.)

```
19      // demonstrate bitwise inclusive OR (|)
20      number1 = 15;
21      unsigned int setBits = 241;
22      puts("\nThe result of combining the following");
23      displayBits(number1);
24      displayBits(setBits);
25      puts("using the bitwise inclusive OR operator | is");
26      displayBits(number1 | setBits);
27
28      // demonstrate bitwise exclusive OR (^)
29      number1 = 139;
30      unsigned int number2 = 199;
31      puts("\nThe result of combining the following");
32      displayBits(number1);
33      displayBits(number2);
34      puts("using the bitwise exclusive OR operator ^ is");
35      displayBits(number1 ^ number2);
36
```

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

```
37      // demonstrate bitwise complement (~)
38      number1 = 21845;
39      puts("\nThe one's complement of");
40      displayBits(number1);
41      puts("is");
42      displayBits(~number1);
43   }
44
```

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

```
45  // display bits of an unsigned int value
46  void displayBits(unsigned int value)
47  {
48     // declare displayMask and left shift 31 bits
49     unsigned int displayMask = 1 << 31;
50
51     printf("%10u = ", value);
52
53     // loop through bits
54     for (unsigned int c = 1; c <= 32; ++c) {
55        putchar(value & displayMask ? '1' : '0');
56        value <<= 1; // shift value left by 1
57
58        if (c % 8 == 0) { // output a space after 8 bits
59           putchar(' ');
60        }
61     }
62
63     putchar('\n');
64  }
```

Fig. 10.9 | Using the 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.

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators (Cont.)

- In Fig. 10.9, integer variable `number1` is assigned value `65535` (`00000000 00000000 11111111 11111111`) and variable `mask` is assigned the value `1` (`00000000 00000000 00000000 00000001`).
- When `number1` and `mask` are combined using the bitwise AND operator (`&`) in the expression `number1 & mask`, 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.

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators (Cont.)

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

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

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

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators (Cont.)

- The *bitwise exclusive OR operator (^)* 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`)
- When these variables are combined with the *bitwise exclusive OR operator* in the expression `number1 ^ number2`, 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 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0

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

10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators (Cont.)

- The bitwise complement operator (~) 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`)
- When the expression `~number1` is evaluated, the result is `11111111 11111111 10101010 10101010`.

10.9.4 Using the Bitwise Left- and Right-Shift Operators

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

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

void displayBits(unsigned int value); // prototype

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

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

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

```

Fig. 10.13 | Using the bitwise shift operators. (Part 1 of 3.)

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

Fig. 10.13 | Using the 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 | Using the bitwise shift operators. (Part 3 of 3.)

10.9.4 Using the Bitwise Left- and Right-Shift 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.
- Bits vacated to the right are replaced with `0`s; `1`s shifted off the left are lost.
- In Fig. 10.13, variable `number1` is assigned the value `960` (`00000000 00000000 00000011 11000000`)
- The result of left shifting variable `number1` 8 bits in the expression `number1 << 8` is `49152` (`00000000 00000011 11000000 00000000`).

10.9.4 Using the Bitwise Left- and Right-Shift Operators (Cont.)

- The *right-shift operator (>>)* 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 int` causes the vacated bits at the left to be replaced by `0`s; `1`s shifted off the right are lost.
- In Fig. 10.13, the result of right shifting `number1` in the expression `number1 >> 8` is `3` (`00000000 00000000 00000000 00000011`).

Common Programming Error 10.11

The result of right or left 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 10.8

The result of right shifting a negative number is implementation defined.

10.9.5 Bitwise Assignment Operators

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

Fig. 10.14 | The bitwise assignment operators.

10.9 Bitwise Operators (Cont.)

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

Operator	Associativity	Type
`()` `[]` `.` `->` `++` *(postfix)* `--` *(postfix)*	left to right	highest
`+` `-` `++` `--` `!` `&` `*` `~` `sizeof` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	shifting
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&`	left to right	bitwise AND
`^`	left to right	bitwise XOR
`|`	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 int` 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 int`.

10.10 Bit Fields (Cont.)

- Consider the following structure definition:

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

which contains three `unsigned int` 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 int` 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

10.10 Bit Fields (Cont.)

- Function `fillDeck` inserts the 52 cards in the `deck` array and function `deal` 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's 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 int a : 13;
       unsignedint    : 19;
       unsigned int 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.
    ```

```c
// Fig. 10.16: fig10_16.c
// Representing cards with bit fields in a struct
#include <stdio.h>
#define CARDS 52

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

typedef struct bitCard Card; // new type name for struct bitCard

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

int main(void)
{
   Card deck[CARDS]; // create array of Cards

   fillDeck(deck);

```

Fig. 10.16 | Representing cards with bit fields in a struct. (Part 1 of 3.)

```
24      puts("Card values 0-12 correspond to Ace through King");
25      puts("Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades");
26      puts("Color values 0-1 correspond to red and black\n");
27      deal(deck);
28   }
29
30   // initialize Cards
31   void fillDeck(Card * const wDeck)
32   {
33      // loop through wDeck
34      for (size_t i = 0; i < CARDS; ++i) {
35         wDeck[i].face = i % (CARDS / 4);
36         wDeck[i].suit = i / (CARDS / 4);
37         wDeck[i].color = i / (CARDS / 2);
38      }
39   }
40
```

Fig. 10.16 | Representing cards with bit fields in a struct. (Part 2 of 3.)

```
41  // output cards in two-column format; cards 0-25 indexed with
42  // k1 (column 1); cards 26-51 indexed with k2 (column 2)
43  void deal(const Card * const wDeck)
44  {
45     printf("%-6s%-6s%-15s%-6s%-6s%s\n", "Card", "Suit", "Color",
46        "Card", "Suit", "Color");
47
48     // loop through wDeck
49     for (size_t k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2) {
50        printf("Card:%3d  Suit:%2d  Color:%2d   ",
51           wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
52        printf("Card:%3d  Suit:%2d  Color:%2d\n",
53           wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
54     }
55  }
```

Fig. 10.16 | Representing cards with bit fields in a struct. (Part 3 of 3.)

```
Card values 0-12 correspond to Ace through King
Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades
Color values 0-1 correspond to red and black

Card  Suit  Color          Card  Suit  Color
0     0     0              0     2     1
1     0     0              1     2     1
2     0     0              2     2     1
3     0     0              3     2     1
4     0     0              4     2     1
5     0     0              5     2     1
6     0     0              6     2     1
7     0     0              7     2     1
8     0     0              8     2     1
9     0     0              9     2     1
10    0     0              10    2     1
11    0     0              11    2     1
12    0     0              12    2     1
0     1     0              0     3     1
1     1     0              1     3     1
2     1     0              2     3     1
3     1     0              3     3     1
4     1     0              4     3     1
5     1     0              5     3     1
6     1     0              6     3     1
7     1     0              7     3     1
8     1     0              8     3     1
9     1     0              9     3     1
10    1     0              10    3     1
11    1     0              11    3     1
12    1     0              12    3     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 int a : 13;
   unsigned int   : 0;
   unsigned int 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.

Performance Tip 10.2

Bit fields help reduce the amount of memory a program needs.

Portability Tip 10.9

Bit-field manipulations are machine dependent.

Common Programming Error 10.12

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 10.13

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 10.3

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.

10.11 Enumeration Constants

- An enumeration (discussed briefly in Section 5.11), 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};
    ```
- Because 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.
    ```

10.11 Enumeration Constants (Cont.)

- 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 `""`.
- You could set `monthName[0]` to a value such as `***ERROR***` to indicate that a logic error occurred.

Common Programming Error 10.14

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

Good Programming Practice 10.5

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

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

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

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

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

Fig. 10.18 | Using an enumeration. (Part 1 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.)

10.12 Anonymous Structures and Unions

- C11 supports anonymous `structs` and `unions` that can be nested in named `structs` and `unions`
- Members in a nested anonymous `struct` and `union` are considered to be members of the enclosing type and can be accessed directly through an object of the enclosing type
- Example
  - ```
    struct MyStruct {
       int member1;
       int member2;

       struct {
          int nestedMember1;
          int nestedMember2;
       };
    };
    ```
  - For a variable `myStruct` of type `struct MyStruct`, you can access the members as:
    - ```
      myStruct.member1
      myStruct.member2
      myStruct.nestedMember1
      myStruct.nestedMember2
      ```

10.13 Secure C Programming

- Various CERT guidelines and rules apply to this chapter's topics. For more information on each, visit `www.securecoding.cert.org`.

`struct`

- The boundary alignment requirements for `struct` members may result in extra bytes containing undefined data for each `struct` variable you create. Each of the following guidelines is related to this issue:

10.13 Secure C Programming (Cont.)

- EXP03-C: Because of *boundary alignment* requirements, the size of a `struct` variable is not necessarily the sum of its members' sizes. Always use `sizeof` to determine the number of bytes in a `struct` variable. As you'll see, we use this technique to manipulate fixed-length records that are written to and read from files in Chapter 11, and to create so-called dynamic data structures in Chapter 12.
- EXP04-C: `struct` variables cannot be compared for equality or inequality, because they might contain bytes of undefined data. Therefore, you must compare their individual members.

10.13 Secure C Programming (Cont.)

- DCL39-C: In a `struct` variable, the undefined extra bytes could contain secure data—left over from prior use of those memory locations—that should not be accessible. This CERT guideline discusses compiler-specific mechanisms for packing the data to eliminate these extra bytes.

typedef

- DCL05-C: Complex type declarations, such as those for function pointers can be difficult to read. You should use `typedef` to create self-documenting type names that make your programs more readable.

Bit Manipulation

- INT02-C: As a result of the integer promotion rules (discussed in Section 5.6), performing bitwise operations on integer types smaller than int can lead to unexpected results. Explicit casts are required to ensure correct results.
- INT13-C: Some bitwise operations on *signed* integer types are *implementation defined*—this means that the operations may have different results across C compilers. For this reason, *unsigned* integer types should be used with the bitwise operators.

10.13 Secure C Programming (Cont.)

- EXP17-C: The logical operators `&&` and `||` are frequently confused with the bitwise operators `&` and `|`, respectively. Using `&` and `|` in the condition of a conditional expression (`?:`) can lead to unexpected behavior, because the `&` and `|` operators do not use short-circuit evaluation.

enum

- INT09-C: Allowing multiple enumeration constants to have the *same* value can result in difficult-to-find logic errors. In most cases, an `enum`'s enumeration constants should each have *unique values* to help prevent such logic errors