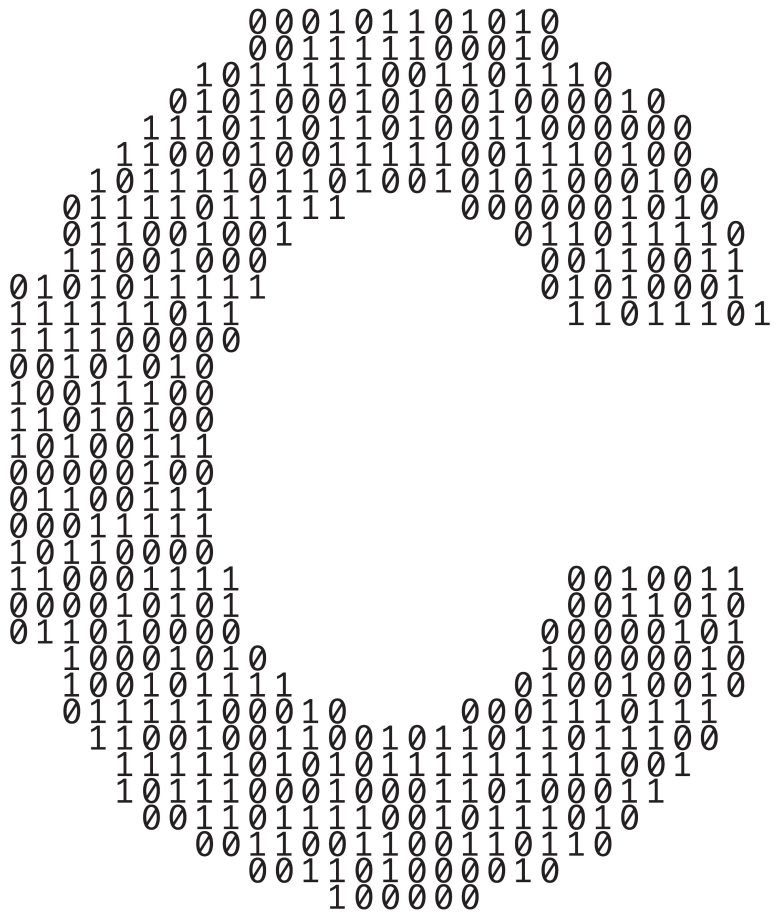
Structures, Unions, Bit Manipulation and Enumerations

10



Objectives

In this chapter, you'll:

- Create and use `structs`, `unions` and `enums`.
- Understand self-referential `structs`.
- 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 `typedefs` to create aliases for existing type names.
- Learn the operations that can be performed on `unions`.
- Initialize `unions`.
- Manipulate integer data with the bitwise operators.
- Create bit fields for storing data compactly.
- Use `enum` constants.
- Consider the security issues of working with `structs`, bit manipulation and `enums`.

Outline

10.1 Introduction

Structures are collections of related variables under one name, known as **aggregates** in the C standard. Structures may contain many variables of different types. That's in contrast to arrays, which contain only elements of the same type. Here, we'll discuss:

- `typedef`s—for creating *aliases* for previously defined data types.
- `union`s—similar to structures, but with members that *share* the *same* storage.
- 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.

In Chapters 11 and 12, you'll see that

- structures commonly define records to be stored in files, and
- pointers and structures facilitate forming data structures such as linked lists, queues, stacks and trees.

Self Check

1 *(Fill-In)* _____ are similar to structures, but with members that share the same storage space.
Answer: `unions`.

2 *(Fill-In)* _____ are sets of integer constants represented by identifiers.
Answer: Enumerations.

10.2 Structure Definitions

Structures are **derived data types**—they're constructed using objects of other types. The keyword `struct` introduces a structure definition, as in

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

The identifier `card` is the **structure tag**, which you use with `struct` to declare variables of the **structure type**—e.g., `struct card`. Variables declared within a `struct`'s braces are the structure's **members**. A `struct`'s members must have unique names, though separate structure types may contain members of the same name without conflict. Each structure definition ends with a semicolon.

The `struct card` definition contains `const char *` members `face` and `suit`. Structure members can be `const` or non-`const` primitive-type variables (e.g., `int`s, `double`s, etc.) or aggregates, such as arrays or other `struct`-type objects. Chapter 6 showed that an array's elements all have the same type. Structure members, however, can be of different types. For example, the following `struct` contains `char` array members for an employee's first and last names, an `int` member for the employee's age and a `double` member for the employee's hourly salary:

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

10.2.1 Self-Referential Structures

A `struct` type may not contain a variable of its own `struct` type (which is a compilation error), but it may contain a pointer to that `struct` type. For example, the updated `struct employee` below contains a pointer to the employee's manager, which would be another `struct employee` object:


```
struct employee {
   char firstName[20];
   char lastName[20];
   unsigned int age;
   double hourlySalary;
   struct employee *managerPtr; // pointer
};
```

A structure containing a member that's a pointer to the *same* `struct` type is a **self-referential structure**. Self-referential structures are used to build linked data structures.

10.2.2 Defining Variables of Structure Types

A structure definition does not reserve any space in memory. Rather, it creates a new data type you can use to define variables. It's like a blueprint showing how to build instances of that `struct`. The following statements reserve memory for variables using the type `struct card`:

```
struct card myCard;
struct card deck[52];
struct card *cardPtr;
```

Variable `myCard` is a `struct card` object, array `deck` consists of 52 `struct card` objects, and `cardPtr` is a pointer to a `struct card` object.

Variables of a given structure type may also be defined by placing a comma-separated list of variable names between the `struct`'s closing brace and terminating semicolon. For example, you can incorporate the preceding definitions into the `struct card` definition:

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

10.2.3 Structure Tag Names

The structure tag name is optional. If a structure definition does not specify a tag name, you must define any variables of the type, as shown in the preceding code snippet. Always provide a structure tag name so you can declare new variables of that type later.

10.2.4 Operations That Can Be Performed on Structures

You can perform the following operations on `structs`:

- assigning one `struct` variable to another of the *same* type (Section 10.7)—for a pointer member, this copies only the address stored in the pointer,
- taking the address (`&`) of a `struct` variable (Section 10.4),
- accessing a `struct` variable's members (Section 10.4),
- using the `sizeof` operator to determine a `struct` variable's size, and
- zero initializing a struct variable in its definition, as in

```
struct card myCard = {};
```

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

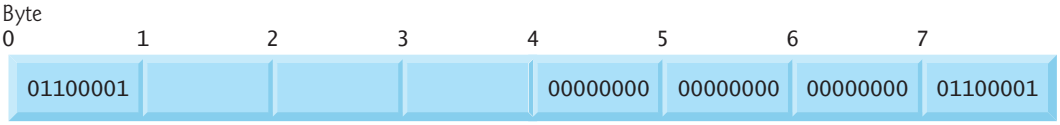
Comparing Structure Objects Is Not Allowed

Structures may *not* be compared using operators `==` and `!=`, because structure members may not be stored in consecutive bytes of memory. Sometimes there are "holes" in a structure because computers store some data types only on certain memory boundaries, such as half-word, word or double-word boundaries. This is machine-dependent. A word is a memory unit used to store data in a computer, usually four bytes or eight bytes.

Consider the following structure definition, which also defines variables `sample1` and `sample2`:

```
struct example {
   char c;
   int i;
} sample1, sample2;
```

A computer with four-byte words might require that each `struct example` member be aligned on a word boundary, i.e., at the beginning of a word. The following diagram shows a possible memory alignment for a `struct example` variable that has been assigned the character `'a'` and the integer `97`. We show the bit representations here.



If each member is stored beginning at a word boundary, each `struct example` variable has a three-byte hole in bytes `1`–`3`. The hole's value is *unspecified*. Even if `sample1`'s and `sample2`'s member values are equal, the holes are not likely to contain identical values, so the structures are not necessarily equal. Data type sizes and memory alignment considerations are machine-dependent. SE

Self Check

1 *(Multiple Choice)* Consider the `struct name` definition:

```
struct name {
   const char *first;
   const char *last;
};
```

Which of the following statements a), b) or c) is *false*?

a) Keyword `struct` introduces the structure definition.
b) The structure tag `name` can be used with `struct` to declare variables of the structure type.
c) Variables declared within a `struct`'s braces are the structure's members, which must have unique names.
d) All of the above statements are *true*.

Answer: d.

2 *(Multiple Choice)* Which of the following a), b) or c) is not a valid operation that may be performed on a structure?

a) Assigning `struct` variables to `struct` variables of the same type.
b) Dereferencing a `struct` variable.
c) Accessing a `struct` variable's members and using `sizeof` to determine the size of a `struct` variable.
d) All of the above statements are *true*.

Answer: b) is not valid. You cannot dereference a `struct` because it's not a pointer, but you can take a `struct`'s address with `&`.

10.3 Initializing Structures

Like arrays, you can initialize a `struct` variable via an initializer list. For example, the following statement creates variable `myCard` using type `struct card` (Section 10.2) and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`:

```
struct card myCard = {"Three", "Hearts"};
```

If there are fewer initializers than members, the remaining members are automatically initialized to `0` or `NULL` (for pointer members). 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. You may also assign structure variables to other structure variables of the same type or assign values to individual structure members.

Self Check

1 *(True/False)* If there are fewer initializers in the list than members in the structure, the remaining members are not initialized.
Answer: *False*. Actually, they're initialized to `0` (or `NULL` if the member is a pointer).

2 *(True/False)* You may assign structure variables to other structure variables of the same type or assign values to individual structure members.
Answer: *True*.

10.4 Accessing Structure Members with . and ->

You can access structure members with:

- the **structure member operator** (`.`), or dot operator, and
- the **structure pointer operator** (`->`), or **arrow operator**.

Structure Member Operator (.)

The structure member operator accesses a structure member via a structure variable name. For example, using the structure variable `myCard` from Section 10.3, we can print the `suit` member with the statement:

```
printf("%s", myCard.suit); // displays Hearts
```

Structure Pointer Operator (->)

You can access a structure member via a pointer to the structure using the structure pointer operator—a minus (-) sign and a greater than (>) sign with no intervening spaces. If the pointer `cardPtr` points to the `struct card` object `myCard` we defined earlier, we can print its member `suit` with the statement:

```
printf("%s", cardPtr->suit); // displays Hearts
```

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 higher precedence than the pointer dereferencing operator (`*`). The structure pointer operator and

structure member operator have the highest precedence and group from left-to-right, along with parentheses (for calling functions) and brackets ([]) used for array indexing.

Spacing Conventions

Do not put spaces around the -> and . (dot) operators to emphasize that the expressions the operators are contained in are essentially single variable names. Inserting space between the structure pointer operator's - and > or between any other multiple-keystroke operator's components (except ?:) is a syntax error. ⊗ ERR

Demonstrating the Structure Member and Structure Pointer Operators

Figure 10.1 refers to members of structure myCard using the structure member and structure pointer operators. Lines 16–17 assign "Ace" and "Spades" to myCard's members. Line 19 assigns myCard's address to cardPtr. Lines 21–23 display myCard's members using:

- the structure member operator and variable name myCard,
- the structure pointer operator and pointer cardPtr, and
- the structure member operator with dereferenced pointer cardPtr.

```
1  // fig10_01.c
2  // Structure member operator and
3  // structure pointer operator
4  #include <stdio.h>
5
6  // card structure definition
7  struct card {
8     const char *face; // define pointer face
9     const char *suit; // define pointer suit
10 };
11
12 int main(void) {
13    struct card myCard; // define one struct card variable
14
15    // place strings into myCard
16    myCard.face = "Ace";
17    myCard.suit = "Spades";
18
19    struct card *cardPtr = &myCard; // assign myCard's address to cardPtr
20
21    printf("%s of %s\n", myCard.face, myCard.suit);
22    printf("%s of %s\n", cardPtr->face, cardPtr->suit);
23    printf("%s of %s\n", (*cardPtr).face, (*cardPtr).suit);
24 }
```

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

Fig. 10.1 | Structure member operator and structure pointer operator.

Self Check

1 *(Fill-In)* The structure member operator _____ and the structure pointer operator _____ can be used to access structure members.
Answer: . (dot) , ->.

2 *(True/False)* 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 optional.
Answer: *False*. The expressions, as shown, are equivalent. The parentheses *are* needed here because the structure member operator (.) has higher precedence than the pointer dereferencing operator (*).

10.5 Using Structures with Functions

With structures, you can pass to functions:

- individual structure members,
- entire structure objects, or
- pointers to structure objects.

Individual structure members and entire structure objects are passed by value, so functions cannot modify them in the caller. To pass a structure by reference, use the structure object's address. Passing structures by reference is more efficient than passing structures by value, which requires the entire structure to be copied. Arrays of structure objects—like all other arrays—are automatically passed by reference.

PERF

Passing an Array By Value

In Chapter 6, we stated that you can use a structure to pass an array by value. To do so, create a structure with an array member. Structures are passed by value, so its members are passed by value.

Self Check

1 *(Fill-In)* Structure objects and individual structure members are passed to functions by _____.
Answer: value.

2 *(Discussion)* How can you pass an array by value?
Answer: Simply place the array in a structure and pass the structure. Structures normally pass by value.

10.6 typedef

The keyword `typedef` enables you to create synonyms (or aliases) for previously defined types. It's commonly used to create shorter names for `struct` types and simplify declarations of types like function pointers. For example, the following `typedef` defines `Card` as a synonym for type `struct card`:

```
typedef struct card Card;
```

By convention, capitalize the first letter of `typedef` names to emphasize that they're synonyms for other type names.

You can now use `Card` 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` creates an alternate *type name*, which may be used as an *alias* for an existing type name. 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 `Card`s."

Combining `typedef` with `struct` Definitions

Programmers often use `typedef` to define a structure type, so a structure tag is not required. For example, the following definition also creates the structure type `Card`:

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

Synonyms for Built-In Types

Using `typedef`s can help make a program more readable and maintainable. Often, `typedef` is used to create synonyms for built-in 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`. To port the program to another platform, you can simply change the `Integer` `typedef` and recompile the program.

Self Check

1 *(Code)* Write a `typedef` statement that creates for structure type `struct dice` the shorter type name `Dice`.

Answer: `typedef struct dice Dice;`

2 *(True/False)* Creating a new name with `typedef` creates a new type.

Answer: *False*. Creating a new name with `typedef` does *not* create a new type. It simply creates a new *type name*, which may be used as an *alias* for an existing type name.

10.7 Random-Number Simulation Case Study: High-Performance Card Shuffling and Dealing

Figure 10.2 is based on Chapter 7's card shuffling and dealing simulation. This program represents the deck of cards as an array of `Card` `struct`s and uses high-performance shuffling and dealing algorithms.

```c
// fig10_02.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

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

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

// prototypes
void fillDeck(Card * const deck, const char *faces[], const char *suits[]);
void shuffle(Card * const deck);
void deal(const Card * const deck);

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

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

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

   srand(time(NULL)); // randomize

   fillDeck(deck, faces, suits); // load the deck with Cards
   shuffle(deck); // put Cards in random order
   deal(deck); // deal all 52 Cards
}

// place strings into Card structures
void fillDeck(Card * const deck, const char * faces[],
   const char * suits[]) {
   // loop through deck
   for (size_t i = 0; i < CARDS; ++i) {
      deck[i].face = faces[i % FACES];
      deck[i].suit = suits[i / FACES];
   }
}
```

Fig. 10.2 | Card shuffling and dealing program using structures. (Part 1 of 2.)

```
50  // shuffle cards
51  void shuffle(Card * const deck) {
52     // loop through deck randomly swapping Cards
53     for (size_t i = 0; i < CARDS; ++i) {
54        size_t j = rand() % CARDS;
55        Card temp = deck[i];
56        deck[i] = deck[j];
57        deck[j] = temp;
58     }
59  }
60
61  // deal cards
62  void deal(const Card * const deck) {
63     // loop through deck
64     for (size_t i = 0; i < CARDS; ++i) {
65        printf("%5s of %-8s%s", deck[i].face, deck[i].suit,
66           (i + 1) % 4 ? "  " : "\n");
67     }
68  }
```

```
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. 10.2 | Card shuffling and dealing program using structures. (Part 2 of 2.)

Line 35 calls function `fillDeck` (lines 41–48) to initialize the `Card` array in order with `"Ace"` through `"King"` of each suit. Line 36 passes the `Card` array to function `shuffle` (lines 51–59), which implements the high-performance shuffling algorithm. Function `shuffle` takes an array of 52 `Card`s as an argument. The function loops through the 52 `Card`s. For each `Card`, the algorithm chooses a random number between 0 and 51, then swaps the current `Card` and the randomly selected `Card`. The algorithm performs 52 swaps in a single pass of the entire array, and the array of `Card`s is shuffled! This algorithm cannot suffer from indefinite postponement like Chapter 7's shuffling algorithm. The `Card`s were swapped in place in the array, so the high-performance dealing algorithm in function `deal` (lines 62–68) can deal the shuffled `Card`s in only *one* pass of the array.

Related Exercise—Fisher-Yates Shuffling Algorithm

It's recommended that you use an unbiased shuffling algorithm for real card games. Such an algorithm ensures that all possible shuffled card sequences are equally likely to occur. Exercise 10.18 asks you to research the popular unbiased Fisher-Yates shuffling algorithm and use it to reimplement function `shuffle` in Fig. 10.2.

Self Check

1 *(Code)* Rewrite the following code to avoid the separate `typedef` statement:

```
struct name {
   const char *first;
   const char *last;
};

typedef struct name Name;
```

Answer:

```
typedef struct name {
   const char *first;
   const char *last;
} Name;
```

2 *(Code)* Correct the following code, which is supposed to swap elements `i` and `j` of the `deck` array of `Cards`:

```
deck[i] = deck[j];
deck[j] = deck[i];
```

Answer:

```
Card temp = deck[i];
deck[j] = deck[i];
deck[i] = temp;
```

3 *(Discussion)* Why did the card shuffling algorithm we presented in Chapter 7 suffer from indefinite postponement? Why doesn't this chapter's card shuffling algorithm suffer from indefinite postponement?

Answer: Chapter 7's card shuffling and dealing example used a 4-by-13 array to represent the four suits and 13 faces in a deck. The shuffling algorithm used sentinel-controlled looping to place 1–52 (representing dealing order) into randomly selected rows and columns. This loop could execute indefinitely if the randomly selected cells already contain one of these values. This chapter's shuffling algorithm uses counter-controlled iteration to make one pass of a one-dimensional array. The loop iterates once for each card, then terminates, so it cannot suffer from indefinite postponement.

10.8 Unions

Like a structure, a **union** is a derived data type, but its members share the same memory. At different times during program execution, some variables may not be relevant when others are. So, a union shares the space rather than wasting storage on variables that are not in use. A union's members can be of any type. The number of bytes used to store a union must be at least enough to hold its largest member.

In most cases, unions contain two or more items of different types. You can reference only one member—and thus only one type—at a time. It's your responsibility to reference the data with the proper type. Referencing the currently stored data with a variable of the wrong type is a logic error—the result is implementation-dependent. ERR

Union Portability

The amount of memory required to store a union is implementation-dependent. SE Operator `sizeof` will always return a value at least as large as the size in bytes of the union's largest member. Some unions may not port easily among computer systems. Whether a union is portable or not often depends on the memory alignment requirements for a union's member types on a given system. SE

10.8.1 union Declarations

The following `union` has two members—`int x` and `double y`:

```
union number {
   int x;
   double y;
};
```

The `union` definition is normally placed in a header and included in all source files that use the `union` type. As with a `struct` definition, a `union` definition simply creates a new type. It does not reserve any memory until you use the type to create variables. SE

10.8.2 Allowed unions Operations

The operations that can be performed on a `union` are:

- assigning a `union` to another `union` of the same type,
- taking a `union` variable's address (`&`),
- accessing `union` members via the structure member operator (`.`) and the structure pointer operator (`->`), and
- zero-initializing the union.

Two `unions` may not be compared using operators `==` and `!=` for the same reasons that two structures cannot be compared.

10.8.3 Initializing unions in Declarations

You can initialize a `union` in a declaration with a value of the `union`'s first member type. The `union number` in Section 10.8.1 has an `int` as its first member, so we can initialize an object of this type with the following statement:

```
union number value = {10};
```

If you initialize the object with a `double`, as in

```
union number value = {1.43};
```

C will truncate the initializer value's floating-point part—some compilers will issue a warning about this.

10.8.4 Demonstrating unions

Figure 10.3 displays a `union number` variable named `value` (line 12) as both an `int` and a `double`. This program's output is implementation-dependent.

```
1  // fig10_03.c
2  // Displaying the value of a union in both member data types
3  #include <stdio.h>
4
5  // number union definition
6  union number {
7     int x;
8     double y;
9  };
10
11 int main(void) {
12    union number value; // define a union variable
13
14    value.x = 100; // put an int into the union
15    puts("Put 100 in the int member and print both members:");
16    printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);
17
18    value.y = 100.0; // put a double into the same union
19    puts("Put 100.0 in the double member and print both members:");
20    printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);
21 }
```

Microsoft Visual Studio

```
Put 100 in the int member and print both members:
int: 100
double: -92559592117433135502616407313071917486139351398276445610442752.00

Put 100.0 in the double member and print both members:
int: 0
double: 100.00
```

GNU GCC and Apple Xcode

```
Put 100 in the int member and print both members:
int: 100
double: 0.00

Put 100.0 in the double member and print both members:
int: 0
double: 100.00
```

Fig. 10.3 | Displaying the value of a `union` in both member data types.

Self Check

1 ***(Discussion)*** Like a `struct`, a `union` is a derived data type. How is a `union` different from a `struct`?

Answer: A `union` shares its memory among all of its members. Only one member may be stored in a `union` at any time. You must keep track of which member is currently stored.

2 ***(True/False)*** The following `union` definition indicates that `number` is a `union` type with members `int x` and `double y`:

```
union number {
   int x;
   double y;
};
```

For a machine with four-byte `int`s and eight-byte `double`s, the compiler must reserve at least 12 bytes for a variable of this **union** type.

Answer: *False*. Only one of these members is active at a time, so the *union* needs to reserve only as much storage as is needed for the largest member—in this case, eight bytes.

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 eight bits forms a byte—the typical storage unit for a `char` variable. The bitwise operators are used to manipulate the bits of integral operands, both `signed` and `unsigned`, though unsigned integers are typically used. Bitwise data manipulations are machine-dependent. The following table summarizes the bitwise operators. SE

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`. This is often called toggling the bits.

Detailed discussions of each bitwise operator appear in the examples that follow. The examples show the binary representations of the integer operands.

10.9.1 Displaying an Unsigned Integer's Bits

When using the bitwise operators, it's useful to display values in binary[1] to show each operator's precise effects. Figure 10.4 prints an `unsigned int` in its binary representation using eight-bit groups for readability. All the compilers we used to test these examples store `unsigned int`s in 4 bytes (32 bits) of memory.

```
1  // fig10_04.c
2  // Displaying an unsigned int in bits
3  #include <stdio.h>
4
5  void displayBits(unsigned int value); // prototype
6
7  int main(void) {
8     unsigned int x = 0; // variable to hold user input
9
10    printf("%s", "Enter a nonnegative int: ");
11    scanf("%u", &x);
12    displayBits(x);
13 }
14
15 // display bits of an unsigned int value
16 void displayBits(unsigned int value) {
17    // define displayMask and left shift 31 bits
18    unsigned int displayMask = 1 << 31;
19
20    printf("%10u = ", value);
21
22    // loop through bits
23    for (unsigned int c = 1; c <= 32; ++c) {
24       putchar(value & displayMask ? '1' : '0');
25       value <<= 1; // shift value left by 1
26
27       if (c % 8 == 0) { // output space after 8 bits
28          putchar(' ');
29       }
30    }
31
32    putchar('\n');
33 }
```

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

Fig. 10.4 | Displaying an `unsigned int` in bits.

1. See online Appendix E for a detailed explanation of the binary (base-2) number system.

Displaying the Bits of an Integer

Function `displayBits` (lines 16–33) uses the bitwise AND operator to combine variable `value` with the variable `displayMask` (line 24). 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`, line 18 assigns the mask variable `displayMask` the value

```
1 << 31         (10000000 00000000 00000000 00000000)
```

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 24

```
putchar(value & displayMask ? '1' : '0');
```

determines whether to display a `1` or a `0` for the current leftmost bit of `value`. Combining `value` and `displayMask` with `&` "masks off" (hides) all the bits except the high-order bit in `value`—any bit "ANDed" with `0` yields `0`. If the leftmost bit is `1`, `value & displayMask` evaluates to a nonzero (true) value and line 24 displays `1`; otherwise, it displays `0`. Line 25 left shifts the variable `value` one bit with the expression `value <<= 1`. Function `displayBits` repeats these steps for each bit in `value`. Using the logical AND operator (`&&`) for the bitwise AND operator (`&`)—and vice versa—is a logic error. The table below summarizes the results of combining two bits with the bitwise AND operator.

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

10.9.2 Making Function `displayBits` More Generic and Portable

In line 18 of Fig. 10.4, 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 23, we hard-coded the integer `32` to indicate that the loop should iterate 32 times, once for each bit in `value`. We assumed that `unsigned int`s are always stored in 32 bits (four bytes) of memory. Today's popular computers generally use 32-bit- or 64-bit-word hardware architectures. As a C programmer, you'll tend to work across many hardware architectures, and sometimes `unsigned int`s will be stored in smaller or larger numbers of bits.

Figure 10.4 can be made more generic and portable by replacing the integers `31` (line 18) and `32` (line 23) with expressions that calculate these integers, based on the size of an `unsigned int` for a given platform. 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 ints`, these expressions evaluate to `31` and `32`. For 64-bit `unsigned ints`, they evaluate to `63` and `64`.

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

Figure 10.5 demonstrates the bitwise AND, the bitwise inclusive OR, the bitwise exclusive OR and the bitwise complement operators. The program uses function `displayBits` (lines 45–62) to display the `unsigned int` values.

```
1  // fig10_05.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 int value); // prototype
7
8  int main(void) {
9     // demonstrate bitwise AND (&)
10    unsigned int number1 = 65535;
11    unsigned int mask = 1;
12    puts("The result of combining the following");
13    displayBits(number1);
14    displayBits(mask);
15    puts("using the bitwise AND operator & is");
16    displayBits(number1 & mask);
17
18    // demonstrate bitwise inclusive OR (|)
19    number1 = 15;
20    unsigned int setBits = 241;
21    puts("\nThe result of combining the following");
22    displayBits(number1);
23    displayBits(setBits);
24    puts("using the bitwise inclusive OR operator | is");
25    displayBits(number1 | setBits);
26
27    // demonstrate bitwise exclusive OR (^)
28    number1 = 139;
29    unsigned int number2 = 199;
30    puts("\nThe result of combining the following");
31    displayBits(number1);
32    displayBits(number2);
33    puts("using the bitwise exclusive OR operator ^ is");
34    displayBits(number1 ^ number2);
35
```

Fig. 10.5 | Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 2.)

```
36     // demonstrate bitwise complement (~)
37     number1 = 21845;
38     puts("\nThe one's complement of");
39     displayBits(number1);
40     puts("is");
41     displayBits(~number1);
42  }
43
44  // display bits of an unsigned int value
45  void displayBits(unsigned int value) {
46     // declare displayMask and left shift 31 bits
47     unsigned int displayMask = 1 << 31;
48
49     printf("%10u = ", value);
50
51     // loop through bits
52     for (unsigned int c = 1; c <= 32; ++c) {
53        putchar(value & displayMask ? '1' : '0');
54        value <<= 1; // shift value left by 1
55
56        if (c % 8 == 0) { // output a space after 8 bits
57           putchar(' ');
58        }
59     }
60
61     putchar('\n');
62  }
```

```
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. 10.5 | Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 2.)

Bitwise AND Operator (&)

Line 10 assigns the value `65535`

```
00000000 00000000 11111111 11111111
```

to the integer variable `number1`, and line 11 assigns the value `1`

```
00000000 00000000 00000000 00000001
```

to the variable `mask`. When you combine `number1` and `mask` using the bitwise AND operator (`&`) in the expression `number1 & mask` (line 16), the result is

```
00000000 00000000 00000000 00000001
```

All the bits except the low-order bit in `number1` are "masked off" (hidden) by "AND-ing" with variable `mask`.

Bitwise Inclusive OR Operator (|)

The bitwise inclusive OR operator sets specific bits to 1 in an operand. Line 19 assigns `15`

```
00000000 00000000 00000000 00001111
```

to the variable `number1` and line 20 assigns `241`

```
00000000 00000000 00000000 11110001
```

to the variable `setBits`. When you combine `number1` and `setBits` with the bitwise inclusive OR operator in the expression `number1 | setBits` (line 25), the result is `255`

```
00000000 00000000 00000000 11111111
```

The following table 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

Bitwise Exclusive OR Operator (^)

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. Line 28 assigns `number1` the value `139`

```
00000000 00000000 00000000 10001011
```

and line 29 assigns `number2` the value `199`

```
00000000 00000000 00000000 11000111
```

When you combine these variables with the bitwise exclusive OR operator in the expression `number1 ^ number2` (line 34), the result is

```
00000000 00000000 00000000 01001100
```

The following table 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

Bitwise Complement Operator (~)

The bitwise complement operator (~) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1. This is otherwise referred to as "taking the **one's complement** of the value." Line 37 assigns `number1` the value `21845`

```
00000000 00000000 01010101 01010101
```

The expression `~number1` (line 41) toggles all the bits producing

```
11111111 11111111 10101010 10101010
```

10.9.4 Using the Bitwise Left- and Right-Shift Operators

Figure 10.6 demonstrates the left-shift (<<) and right-shift (>>) operators. Again, we use the function `displayBits` to display the `unsigned int` values.

```
1  // fig10_06.c
2  // Using the bitwise shift operators
3  #include <stdio.h>
4
5  void displayBits(unsigned int value); // prototype
6
7  int main(void) {
8     unsigned int number1 = 960; // initialize number1
9
10    // demonstrate bitwise left shift
11    puts("\nThe result of left shifting");
12    displayBits(number1);
13    puts("8 bit positions using the left shift operator << is");
14    displayBits(number1 << 8);
15
16    // demonstrate bitwise right shift
17    puts("\nThe result of right shifting");
18    displayBits(number1);
19    puts("8 bit positions using the right shift operator >> is");
20    displayBits(number1 >> 8);
21 }
22
```

Fig. 10.6 | Using the bitwise shift operators. (Part 1 of 2.)

```
23  // display bits of an unsigned int value
24  void displayBits(unsigned int value) {
25     // declare displayMask and left shift 31 bits
26     unsigned int displayMask = 1 << 31;
27
28     printf("%10u = ", value);
29
30     // loop through bits
31     for (unsigned int c = 1; c <= 32; ++c) {
32        putchar(value & displayMask ? '1' : '0');
33        value <<= 1; // shift value left by 1
34
35        if (c % 8 == 0) { // output a space after 8 bits
36           putchar(' ');
37        }
38     }
39
40     putchar('\n');
41  }
```

```
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. 10.6 | Using the bitwise shift operators. (Part 2 of 2.)

Left-Shift Operator (<<)

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 0s. Bits shifted off the left are lost. Line 8 assigns the variable `number1` the value `960`

```
00000000 00000000 00000011 11000000
```

Left-shifting `number1` eight bits with the expression `number1 << 8` (line 14) results in the value `245760`

```
00000000 00000011 11000000 00000000
```

Right-Shift Operator (>>)

The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Right-shifting an `unsigned int` replaces the vacated bits at the left with 0s. Bits shifted off the right are lost. The result of right-shifting `number1` with the expression `number1 >> 8` (line 20) is `3`

```
00000000 00000000 00000000 00000011
```

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 the left operand. The result of right-shifting a negative number is implementation-defined.

ERR

SE

10.9.5 Bitwise Assignment Operators

Each binary bitwise operator has a corresponding assignment operator. The following table summarizes these **bitwise assignment operators**.

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.

The following table shows in decreasing order the precedence and grouping of the operators introduced to this point in the text.

Operator	Grouping	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
`?:`	left-to-right	conditional
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`	left-to-right	assignment
`,`	left-to-right	comma

Self Check

1 ***(Fill-In)*** Often, the bitwise AND operator is used with an operand called a _____, which is an integer value with specific bits set to `1`. This is used to hide some bits in a value while selecting other bits.

Answer: mask.

2 ***(True/False)*** The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1.
Answer: *False*. Actually, what's described above is the bitwise inclusive OR operator. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bits in each operand are different.

3 ***(Fill-In)*** The bitwise complement operator sets all `0` bits in its operand to `1` in the result and sets all `1` bits to `0`. This is often called _____ the bits.
Answer: toggling.

4 ***(True/False)*** Because of the machine-dependent nature of bitwise manipulations, programs including them might not work correctly or might work differently across systems.
Answer: *True*.

10.10 Bit Fields

You can specify the number of bits in which to store an `unsigned` or signed integral member of a `struct` or `union`. Known as **bit fields**, these enable better memory utilization by storing data in the minimum number of bits required. Bit field members typically are declared as `int` or `unsigned int`.

10.10.1 Defining Bit Fields

The following `struct bitCard`

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

contains three `unsigned int` bit fields—`face`, `suit` and `color`—that can represent a card in a deck of 52 cards. You declare a bit field by following an `unsigned` or signed integral member's name with a colon (`:`) and an integer constant representing the bit field's **width**—the number of bits in which to store the member. The width must be an integer constant 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 four-byte (32-bit) integers.

The preceding structure definition indicates that members `face`, `suit` and `color` are stored in 4 bits, 2 bits and 1 bit, respectively. The number of bits is based on the desired range of values for each member:

- `face` stores values from `0` (Ace) through `12` (King)—4 bits can store values in the range 0–15,
- `suit` stores values from `0` through `3` (`0` = Hearts, `1` = Diamonds, `2` = Clubs, `3` = Spades)—2 bits can store values in the range `0`–`3`, and
- `color` stores either `0` (Red) or `1` (Black)—1 bit can store either `0` or `1`.

10.10.2 Using Bit Fields to Represent a Card's Face, Suit and Color

Figure 10.7 creates the array `deck` containing 52 `struct bitCard` structures in line 19. Function `fillDeck` (lines 30–37) inserts the 52 cards in the `deck` array, and function `deal` (lines 41–49) prints the 52 cards. Notice that bit field members of structures are accessed exactly as any other structure member.

```
1  // fig10_07.c
2  // Representing cards with bit fields in a struct
3  #include <stdio.h>
4  #define CARDS 52
5
6  // bitCard structure definition with bit fields
7  struct bitCard {
8     unsigned int face : 4; // 4 bits; 0-15
9     unsigned int suit : 2; // 2 bits; 0-3
10    unsigned int color : 1; // 1 bit; 0-1
11 };
12
13 typedef struct bitCard Card; // new type name for struct bitCard
14
15 void fillDeck(Card * const deck); // prototype
16 void deal(const Card * const deck); // prototype
17
18 int main(void) {
19    Card deck[CARDS]; // create array of Cards
20
21    fillDeck(deck);
22
23    puts("Card values 0-12 correspond to Ace through King");
24    puts("Suit values 0-3 correspond to Hearts, Diamonds, Clubs and Spades");
25    puts("Color values 0-1 correspond to red and black\n");
26    deal(deck);
27 }
28
29 // initialize Cards
30 void fillDeck(Card * const deck) {
31    // loop through deck
32    for (size_t i = 0; i < CARDS; ++i) {
33       deck[i].face = i % (CARDS / 4);
34       deck[i].suit = i / (CARDS / 4);
35       deck[i].color = i / (CARDS / 2);
36    }
37 }
38
39 // output cards in two-column format; cards 0-25 indexed with
40 // k1 (column 1); cards 26-51 indexed with k2 (column 2)
41 void deal(const Card * const deck) {
42    // loop through deck
43    for (size_t k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2) {
```

Fig. 10.7 | Representing cards with bit fields in a struct. (Part 1 of 2.)

```
44          printf("Card:%3d  Suit:%2d  Color:%2d   ",
45             deck[k1].face, deck[k1].suit, deck[k1].color);
46          printf("Card:%3d  Suit:%2d  Color:%2d\n",
47             deck[k2].face, deck[k2].suit, deck[k2].color);
48       }
49    }
```

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

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. 10.7 | Representing cards with bit fields in a struct. (Part 2 of 2.)

PERF Bit fields can reduce the amount of memory a program needs, but are machine-dependent. 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. PERF This is one of many examples of the kinds of space/time trade-offs that occur in computer science.

Bit fields do not have addresses, so attempting to take the address of a bit field ERR with the `&` operator is an error. Also, using `sizeof` with a bit field is an error.

10.10.3 Unnamed Bit Fields

An **unnamed bit field** is used as **padding** in a `struct`. For example, the definition

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

uses an unnamed 19-bit field as padding. Nothing can be stored in those 19 bits. Member `b` (assuming a four-byte-word computer) is stored in a separate word of memory.

An **unnamed bit field with a zero width** aligns the next bit field on a new storage-unit boundary. For example, the `struct`

```
struct example {
   unsigned int a : 13;
   unsigned int   : 0;
   unsigned int : 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.

Self Check

1 *(Fill-In)* The structure definition

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

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

2 *(Multiple Choice)* Which of the following statements a), b) or c) about Section 10.10.1's `struct bitCard` is *false*?

a) A bit field is declared by following an `unsigned` or signed integral member name with a colon (`:`) and an integer constant representing the bit field's width.
b) The `struct bitCard` 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.
c) The number of bits in a bit field is based on each structure member's desired range of values.
d) All of the above statements are *true*.

Answer: d.

10.11 Enumeration Constants

Section 5.11 introduced the keyword `enum` for defining a set of integer **enumeration constants** represented by identifiers. Values in an `enum` start with `0`, unless specified otherwise, and increment by `1`. For example, the enumeration

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

creates the new type `enum months` in which the identifiers are set to the integers `0` through `11`. To number the months 1 to 12, use:

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

which explicitly sets `JAN` to `1`. The remaining values increment from `1`, resulting in the values `1` through `12`.

The identifiers in any enumeration accessible in a given scope must be unique. Each enumeration constant's value can be set explicitly in the definition by assigning a value to the identifier. Multiple enumeration members can have the same constant value.

ERR ⊗ Assigning a value to an enumeration constant after it's been defined is a syntax error. You should use only uppercase letters in enumeration constant names to make them stand out in a program and as a reminder that enumeration constants are not variables.

Figure 10.8 uses the enumeration variable `month` in a `for` statement to print the months of the year from the array `monthName`. We set `monthName[0]` to the empty string `""` and ignore it in this example.

```
1  // fig10_08.c
2  // Using an enumeration
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
10 int main(void) {
11    // initialize array of pointers
12    const char *monthName[] = { "", "January", "February", "March",
13       "April", "May", "June", "July", "August", "September", "October",
14       "November", "December" };
15
16    // loop through months
17    for (enum months month = JAN; month <= DEC; ++month) {
18       printf("%2d%11s\n", month, monthName[month]);
19    }
20 }
```

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
```

Fig. 10.8 | Using an enumeration. (Part 1 of 2.)

```
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

Fig. 10.8 | Using an enumeration. (Part 2 of 2.)

Self Check

1 *(Code)* The following enumeration creates a new type, `enum days`, in which the identifiers are set to the integers `0` to `6`:

```
enum days {
   MON, TUE, WED, THU, FRI, SAT, SUN
};
```

Rewrite this enumeration to number the days 1 to 7.
Answer:

```
enum days {
   MON = 1, TUE, WED, THU, FRI, SAT, SUN
;
```

2 *(True/False)* Multiple enumeration constants in the same scope can have the same identifier.
Answer: *False*. Actually, the identifiers in any enumeration accessible in the same scope must be unique. Multiple members of an enumeration can have the same constant value.

10.12 Anonymous Structures and Unions

Anonymous `structs` and `unions` can be nested in named `structs` and `unions`. The members in a nested anonymous `struct` or `union` are members of the enclosing `struct` or `union`. They can be accessed directly through an object of the enclosing type. For example, consider the following `struct` declaration:

```
struct myStruct {
   int member1;
   int member2;

   struct { # anonymous struct
      int nestedMember1;
      int nestedMember2;
   }; // end nested struct
}; // end outer struct
```

For a `struct myStruct` variable named `object`, you can access the members as

```
object.member1;
object.member2;
object.nestedMember1;
object.nestedMember2;
```

Self Check

1 ***(True/False)*** The members in a nested anonymous `struct` or `union` are members of the enclosing `struct` or `union`. They can be accessed directly through an object of the enclosing type.
Answer: *True.*

SEC 10.13 Secure C Programming

Various CERT guidelines and rules apply to this chapter's topics. For more information on each, visit `https://wiki.sei.cmu.edu/`.

CERT Guidelines for `structs`

As we discussed in Section 10.2.4, 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:

- EXP03-C: Because of boundary alignment requirements, a `struct` variable's size is *not* necessarily the sum of its members' sizes. Always use `sizeof` to determine a `struct` variable's number of bytes. We'll use this technique to manipulate fixed-length records that are written to and read from files (Chapter 11) and to create custom data structures (Chapter 12).
- EXP04-C: Section 10.2.4 discussed that `struct` variables cannot be compared for equality or inequality because they might contain bytes of undefined data. Therefore, you must compare their individual members.
- 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.

CERT Guideline for `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.

CERT Guidelines for 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.

- EXP46-C: The logical operators `&&` and `||` are frequently confused with the bitwise operators `&` and `|`, respectively. Using `&` and `|` in a conditional expression's condition (`?:`) can lead to unexpected behavior because the `&` and `|` operators do not use short-circuit evaluation.

CERT Guideline for `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.

Self Check

1 ***(Fill-In)*** `struct` variables cannot be compared for equality or inequality, because they might contain bytes of undefined data. Instead, you must _____.
Answer: compare their individual members.

2 ***(True/False)*** 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.
Answer: *True*.

Summary

Section 10.1 Introduction

- **Structures** (p. 536) are collections of related variables under one name. They may contain variables of many different data types.
- Structures are commonly used to define records to be stored in files.
- Pointers and structures can be used to form more complex data structures, such as linked lists, queues, stacks and trees.

Section 10.2 Structure Definitions

- Keyword **`struct`** introduces a structure definition (p. 537).
- The **structure tag** (p. 537) following keyword `struct` names the structure definition. It's used with the keyword `struct` to declare variables of the `struct` type.
- Variables declared within the braces of a `struct` definition are the `struct`'s **members**.
- Members of the same `struct` type must have unique names.
- Each `struct` definition must end with a semicolon.
- `struct` members can have primitive or aggregate data types.
- A `struct` cannot contain an instance of itself but may include a pointer to its type.
- A `struct` containing a member that's a pointer to the same `struct` type is referred to as a **self-referential structure**. Self-referential structures (p. 537) are used to build linked data structures.
- `struct` definitions create new data types that are used to define variables.
- Variables of a given `struct` type can be declared by placing a comma-separated list of variable names between the `struct` definition's closing brace and its ending semicolon.

- If a `struct` definition does not contain a structure tag name, variables of the `struct` type may be declared only in the `struct` definition.
- The only valid operations that may be performed on `structs` are assigning `struct` variables to variables of the same type, taking the address (`&`) of a `struct` variable, accessing the members of a `struct` variable and using the `sizeof` operator to determine the size of a `struct` variable.

Section 10.3 Initializing Structures

- `structs` can be initialized using **initializer lists**.
- If there are fewer initializers in the list than members in the `struct`, the remaining members are automatically initialized to `0` (or `NULL` if the member is a pointer).
- Members of `struct` variables defined outside a function definition are initialized to `0` or `NULL` if they're not explicitly initialized in the external definition.

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

- The **structure member operator** (`.`) and the **structure pointer operator** (`->`) are used to access structure members (p. 540).
- The structure member operator accesses a structure member via a `struct` variable name.
- The structure pointer operator accesses a `struct` member via a pointer to a `struct` object (p. 540).

Section 10.5 Using Structures with Functions

- `struct` members, entire `struct` objects or pointers to `struct` objects may be passed to functions.
- Entire `struct` objects are **passed by value by default**.
- To pass a `struct` object by reference, pass its address. Arrays of `struct` objects are automatically passed by reference.
- To **pass an array by value**, create a `struct` with the array as a member. `structs` are passed by value, so the array is passed by value.

Section 10.6 `typedef`

- The keyword **`typedef`** (p. 542) creates synonyms for previously defined types.
- Names for structure types are often defined with `typedef` to create shorter type names.

Section 10.8 Unions

- A **union** (p. 546) is declared with the keyword **`union`**. Its members share the same storage space.
- A `union`'s members can be of any data type. Operator `sizeof` will always return a value at least as large as the size in bytes of the union's largest member
- Only one `union` member can be referenced at a time. It's your responsibility to access the currently stored member.
- The valid operations on a `union` are assigning a `union` to another 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.
- A `union` may be initialized in a declaration with a value of the first `union` member's type.

Section 10.9 Bitwise Operators

- Computers represent all data internally as sequences of bits with the values `0` or `1`.
- On most systems, a sequence of **8 bits form 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** manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both signed and unsigned). Unsigned integers are normally used.
- The bitwise operators (p. 549) are **bitwise AND** (`&`), **bitwise inclusive OR** (`|`), **bitwise exclusive OR** (`^`), **left shift** (`<<`), **right shift** (`>>`) and **complement** (`~`).
- The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The **bitwise AND operator** (p. 549) sets each bit in the result to 1 if the corresponding bit in both operands is 1. The **bitwise inclusive OR** operator (p. 549) 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** (p. 549) sets each bit in the result to 1 if the corresponding bits in both operands are different.
- The **left-shift operator** (p. 549) 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; bits shifted off the left are lost.
- The **right-shift operator** (p. 549) shifts the bits in 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; bits shifted off the right are lost.
- The **bitwise complement operator** (p. 549) sets all `0` bits in its operand to `1` and all `1` bits to `0` in the result.
- Often, bitwise AND is used with an operand called a **mask** (p. 551)—an integer value with specific bits set to `1`. Masks are used to hide some bits in a value while selecting other bits.
- **`CHAR_BIT`** (p. 551; defined in `<limits.h>`) represents the number of bits in a byte (normally 8). It can be used to make a bit-manipulation program more generic and portable.
- Each binary bitwise operator has a corresponding **bitwise assignment operator** (p. 557).

Section 10.10 Bit Fields

- A **bit field** (p. 558) specifies the number of bits in which an `unsigned` or signed integral member of a structure or union is stored.
- 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 (p. 558). The constant must be an integer between 0 and the total number of bits used to store an `int` on your system, inclusive.
- Bit-field members of structures are accessed exactly as any other structure member.
- It's possible to specify an **unnamed bit field** (p. 561) to be used as **padding** in a structure (p. 561).
- An **unnamed bit field with a zero width** (p. 561) aligns the next bit field on a new storage-unit boundary.

Section 10.11 Enumeration Constants

- An **`enum`** defines a set of integer constants represented by identifiers (p. 561). Values in an `enum` start with `0`, unless specified otherwise, and are incremented by `1`.
- The identifiers in an `enum` must be unique.
- The value of an `enum` constant can be set explicitly via assignment in the `enum` definition.

Self-Review Exercises

10.1 Fill-In the blanks in each of the following:

a) A(n) _____ is a collection of related variables under one name.
b) A(n) _____ is a collection of variables under one name in which the variables share the same memory.
c) In an expression using the _____ operator, bits are set to 1 if the corresponding bits in each operand are 1. Otherwise, the bits are set to zero.
d) The variables declared in a structure definition are called its _____ .
e) In an expression using the _____ operator, bits are set to 1 if at least one of the corresponding bits in either operand is 1. Otherwise, the bits are set to 0.
f) Keyword _____ introduces a structure declaration.
g) Keyword _____ creates a synonym for a previously defined data type.
h) In an expression using the _____ operator, bits are set to 1 if exactly one of the corresponding bits in either operand is 1. Otherwise, the bits are set to 0.
i) The bitwise AND operator (`&`) is often used to _____ bits—that is, to select certain bits while zeroing others.
j) Keyword _____ is used to introduce a union definition.
k) The name of the structure is referred to as the structure _____.
l) You access a structure member with the _____ or _____ operators.
m) The _____ and _____ operators shift the bits of a value left or right.
n) A(n) _____ is a set of integers represented by identifiers.

10.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) `structs` may contain variables of only one data type.
b) Two `unions` can be compared (using `==`) to determine whether they're equal.
c) The tag name of a `struct` is optional.
d) Members of different `structs` must have unique names.
e) Keyword `typedef` is used to define new data types.
f) `structs` are always passed to functions by reference.
g) `structs` may not be compared by using operators `==` and `!=`.

10.3 Write code to accomplish each of the following:

a) Define a `struct` called `part` containing `unsigned int` variable `partNumber` and `char` array `partName` with values that may be as long as 25 characters (including the terminating null character).
b) Define `Part` to be a synonym for the type `struct part`.
c) Use `Part` to declare variable `a` to be of type `struct part`, array `b[10]` to be of type `struct part` and variable `ptr` to be of type pointer to `struct part`.
d) Read a part number and a part name from the keyboard into the individual members of variable `a`.
e) Assign the member values of variable `a` to element 3 of array `b`.
f) Assign the address of array `b` to the pointer variable `ptr`.
g) Print the member values of element 3 of array `b` using the variable `ptr` and the structure pointer operator to refer to the members.

10.4 Find the error in each of the following:

a) Assume that `struct card` contains two `const char *` pointers named `face` and `suit`. Also, the variable `c` is a `struct card`, and the variable `cPtr` is a pointer to `struct card`. Variable `cPtr` has been assigned the address of `c`.

```
printf("%s\n", *cPtr->face);
```

b) Assume that `struct card` contains two `const char *` pointers named `face` and `suit`. Also, the array `hearts[13]` is an array of type `struct card`. The following statement should print the member `face` of array element 10.

```
printf("%s\n", hearts.face);
```

c)
```
union values {
   char w;
   float x;
   double y;
};
union values v = {1.27};
```

d)
```
struct person {
   char lastName[15];
   char firstName[15];
   unsigned int age;
}
```

e) Assume `struct person` has been defined as in part (d) but with the appropriate correction.

```
person d;
```

f) Assume variable `p` has type `struct person` and the variable `c` is a `struct card`.

```
p = c;
```

Answers to Self-Review Exercises

10.1 a) structure. b) union. c) bitwise AND (`&`). d) members. e) bitwise inclusive OR (`|`). f) `struct`. g) `typedef`. h) bitwise exclusive OR (`^`). i) mask. j) `union`. k) tag name. l) structure member, structure pointer. m) left-shift (`<<`), right-shift (`>>`). n) enumeration.

10.2 See the answers below:

a) *False*. A structure can contain variables of many data types.

b) *False*. Unions cannot be compared because there might be bytes of undefined data with different values in union variables that are otherwise identical.

c) *True*.

d) *False*. The members of separate structures can have the same names, but the members of a given structure must have unique names.

e) *False*. Keyword `typedef` is used to define new names (synonyms) for previously defined data types.
f) *False*. Structures are always passed to functions by value.
g) *True*, because of alignment problems.

10.3 See the answers below:

a)
```
struct part {
   unsigned int partNumber;
   char partName[25];
};
```
b)
```
typedef struct part Part;
```
c)
```
Part a, b[10], *ptr;
```
d)
```
scanf("%d%24s", &a.partNumber, a.partName);
```
e)
```
b[3] = a;
```
f)
```
ptr = b;
```
g)
```
printf("%d %s\n", (ptr + 3)->partNumber, (ptr + 3)->partName);
```

10.4 See the answers below:

a) The parentheses that should enclose `*cPtr` have been omitted, causing the order of evaluation of the expression to be incorrect. The expression should be
```
cPtr->face
```
or
```
(*cPtr).face
```
b) The array index is missing. The expression should be `hearts[10].face`.
c) A union can be initialized only with a value that has the same type as the union's first member.
d) A semicolon is required to end a structure definition.
e) Keyword `struct` was omitted from the variable declaration. The declaration should be
```
struct person d;
```
f) Variables of different structure types cannot be assigned to one another.

Exercises

10.5 Provide the definition for each of the following structures and unions:

a) `struct inventory` containing character array `partName[30]`, integer `partNumber`, floating-point `price`, integer `stock` and integer `reorder`.
b) `union data` containing `char c`, `short s`, `long b`, `float f` and `double d`.
c) A `struct` called `address` that contains character arrays `streetAddress[25]`, `city[20]`, `state[3]` and `zipCode[6]`.
d) `struct student` that contains arrays `firstName[15]` and `lastName[15]` and variable `homeAddress` of type `struct address` from part (c).
e) `struct test` containing sixteen-bit fields of one bit each. The names of the bit fields are the letters `a` to `p`.

10.6 Given the following `struct` and variable definitions:

```
struct customer {
   char lastName[15];
   char firstName[15];
   unsigned int customerNumber;

   struct {
      char phoneNumber[11];
      char address[50];
      char city[15];
      char state[3];
      char zipCode[6];
   } personal;
} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

write an expression that accesses the `struct` members in each of the following parts:

a) Member `lastName` of `struct customerRecord`.
b) Member `lastName` of the `struct` pointed to by `customerPtr`.
c) Member `firstName` of `struct customerRecord`.
d) Member `firstName` of the `struct` pointed to by `customerPtr`.
e) Member `customerNumber` of `struct customerRecord`.
f) Member `customerNumber` of the `struct` pointed to by `customerPtr`.
g) Member `phoneNumber` of member `personal` of `struct customerRecord`.
h) Member `phoneNumber` of member `personal` of the `struct` pointed to by `customerPtr`.
i) Member `address` of member `personal` of `struct customerRecord`.
j) Member `address` of member `personal` of the `struct` pointed to by `customerPtr`.
k) Member `city` of member `personal` of `struct customerRecord`.
l) Member `city` of member `personal` of the `struct` pointed to by `customerPtr`.
m) Member `state` of member `personal` of `struct customerRecord`.
n) Member `state` of member `personal` of the `struct` pointed to by `customerPtr`.
o) Member `zipCode` of member `personal` of `struct customerRecord`.
p) Member `zipCode` of member `personal` of the `struct` pointed to by `customerPtr`.

10.7 ***(Card Shuffling and Dealing Modification)*** Modify Figure 10.7 to shuffle the cards using a high-performance shuffle (as shown in Fig. 10.2). Print the resulting deck in a two-column format that uses the face and suit names. Precede each card with its color.

10.8 ***(Using Unions)*** Create `union integer` with members `char c`, `short s`, `int i` and `long b`. Write a program that inputs values of type `char`, `short`, `int` and `long` and stores the values in union variables of type `union integer`. Each `union` variable should be printed as a `char`, a `short`, an `int` and a `long`. Do the values always print correctly?

10.9 *(Using Unions)* Create `union floatingPoint` with members `float f`, `double d` and `long double x`. Write a program that inputs values of type `float`, `double` and `long double` and stores the values in `union` variables of type `union floatingPoint`. Each `union` variable should be printed as a `float`, a `double` and a `long double`. Do the values always print correctly?

10.10 *(Right-Shifting Integers)* Write a program that right-shifts an integer variable 4 bits. The program should print the integer in bits before and after the shift operation. Does your system place `0`s or `1`s in the vacated bits?

10.11 *(Left-Shifting Integers)* Left-shifting an `unsigned int` by 1 bit is equivalent to multiplying the value by 2. Write function `power2` that takes two integer arguments `number` and `pow` and calculates

```
number * 2pow
```

Use the shift operator to calculate the result. Print the values as integers and as bits.

10.12 *(Packing Characters into an Integer)* The left-shift operator can be used to pack four character values into a four-byte `unsigned int` variable. Write a program that inputs four characters from the keyboard and passes them to function `packCharacters`. To pack four characters into an `unsigned int` variable, assign the first character to the `unsigned int` variable, shift the `unsigned int` variable left by 8 bit positions and combine the `unsigned` variable with the second character using the bitwise inclusive OR operator. Repeat this process for the third and fourth characters. Print the characters in their bit format before and after they're packed into the `unsigned int` to prove that the characters are, in fact, packed correctly in the `unsigned int` variable.

10.13 *(Unpacking Characters from an Integer)* Using the right-shift operator, the bitwise AND operator and a mask, write function `unpackCharacters` that takes the `unsigned int` from Exercise 10.12 and unpacks it into four characters. To unpack characters from a four-byte `unsigned int`, combine the `unsigned int` with the mask `4278190080` (`11111111 00000000 00000000 00000000`) and right-shift the result 8 bits. Assign the resulting value to a `char` variable. Then combine the `unsigned int` with the mask `16711680` (`00000000 11111111 00000000 00000000`). Assign the result to another `char` variable. Continue this process with the masks `65280` and `255`. Print the `unsigned int` in bits before it's unpacked, then print the characters in bits to confirm that they were unpacked correctly.

10.14 *(Reversing the Order of an Integer's Bits)* Write a program that reverses the order of the bits in an `unsigned int` value. The program should input the value from the user and call function `reverseBits` to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.

10.15 *(Portable `displayBits` Function)* Modify function `displayBits` of Fig. 10.4 so it's portable between systems using two-byte integers and systems using four-byte

integers. [*Hint:* Use the `sizeof` operator to determine the size of an integer on a particular machine.]

10.16 ***(What's the Value of X?)*** The following program uses function `multiple` to determine if the integer entered from the keyboard is a multiple of some integer `X`. Examine the function `multiple`, then determine `X`'s value.

```
1  // ex10_16.c
2  // This program determines whether a value is a multiple of X.
3  #include <stdio.h>
4
5  int multiple(int num); // prototype
6
7  int main(void) {
8     int y; // y will hold an integer entered by the user
9
10    puts("Enter an integer between 1 and 32000: ");
11    scanf("%d", &y);
12
13    // if y is a multiple of X
14    if (multiple(y)) {
15       printf("%d is a multiple of X\n", y);
16    }
17    else {
18       printf("%d is not a multiple of X\n", y);
19    }
20 }
21
22 // determine whether num is a multiple of X
23 int multiple(int num) {
24    int mask = 1; // initialize mask
25    int mult = 1; // initialize mult
26
27    for (int i = 1; i <= 10; ++i, mask <<= 1) {
28       if ((num & mask) != 0) {
29          mult = 0;
30          break;
31       }
32    }
33
34    return mult;
35 }
```

10.17 What does the following program do?

```
1  // ex10_17.c
2  #include <stdio.h>
3
4  int mystery(unsigned int bits); // prototype
5
6  int main(void) {
7     unsigned int x; // x will hold an integer entered by the user
8
```

```
   puts("Enter an integer: ");
   scanf("%u", &x);

   printf("The result is %d\n", mystery(x));
}

// What does this function do?
int mystery(unsigned int bits) {
   unsigned int mask = 1 << 31; // initialize mask
   unsigned int total = 0; // initialize total

   for (unsigned int i = 1; i <= 32; ++i, bits <<= 1) {
      if ((bits & mask) == mask) {
         ++total;
      }
   }

   return !(total % 2) ? 1 : 0;
}
```

10.18 ***(Fisher-Yates Shuffling Algorithm)*** Research the Fisher-Yates shuffling algorithm online, then use it to reimplement the `shuffle` function in Fig. 10.2.

Special Section: Raylib Game-Programming Case Studies

You're about to begin an exciting and challenging journey into the worlds of graphics, animation, multimedia and game development with the free, open-source, cross-platform raylib game programming library.[2,3] The library supports Windows, macOS, Linux and several other platforms, including Android, Raspberry Pi and the web. Raylib is a C library, but it can be used with C++, C#, Java, JavaScript, Python and many other programming languages.[4]

In this Special Section's first three case studies, you'll study two games and a simulation that we created to help you learn raylib fundamentals:

- In Exercise 10.19, you'll study our completely coded **SpotOn** game, which tests your reflexes by requiring you to click fast-moving spots before they disappear. With each new game level, the spots move even faster, making the game more challenging.
- In Exercise 10.20, you'll study our completely coded **Cannon** game, which challenges you to destroy nine moving targets before a time limit expires. A moving blocker makes the game more challenging.
- In Exercise 10.21, you'll use a dynamic visualization to make the law of large numbers "come alive." You'll study our completely coded die-rolling simulation that displays an animated bar chart. As the simulation rolls the die, it

2. Raylib is Copyright ©2013-2020 Ramon Santamaria (`@raysan5`).
3. "raylib." Accessed November 14, 2020. `https://www.raylib.com`.
4. "raylib bindings." Accessed December 14, 2020. `https://github.com/raysan5/raylib/blob/master/BINDINGS.md`.

updates the frequencies in an array. Then, it displays each die face's frequency, its percentage of the total rolls and a bar representing the frequency's magnitude. For a six-sided die, the values 1 through 6 should each occur with "equal likelihood"—the probability of each is $1/6^{th}$ or 16.67%. If we rolled a die 6000 times, we'd expect about 1000 of each face. Like coin tossing, die rolling is random, so some faces could occur fewer or more than 1000 times. As the number of die rolls increases, you'll watch the frequencies approach 16.67% and the bars in the bar chart become nearly identical in length, confirming the law of large numbers.

These games and simulations use many raylib capabilities—shapes, text, colors, sounds, animation, collision detection and handling user-input events (such as mouse clicks and keystrokes). Each exercise suggests improvements you can make to our code.

Studying Our Complete Code Solutions

A key aspect of becoming a professional programmer is reading and understanding lots of other people's code. You'll frequently visit sites like `GitHub.com` looking for open-source code that you can incorporate into your own projects. For these first three raylib case studies, we provide fully coded solutions in the `raylib` subfolder with the chapter's example code that you download from

```
https://deitel.com/c-how-to-program-9-e
```

Each source-code file includes extensive comments that:

- overview the code's top-level functions,
- list the raylib functions we use, and
- provide details you need to understand how each program works.

You should compile, run and play with each and carefully study our code. This will be challenging but rewarding. You'll work with the cool, open-source raylib package, taking a nice leap into computer graphics and game programming. You'll then have a good foundation for attempting our suggested code modifications and other game-programming exercises.

Raylib Sample Code

The raylib development team provides many **C programming demos** at

```
https://www.raylib.com/examples.html
```

and **sample games** at

```
https://www.raylib.com/games.html
```

with complete source code. Consider studying the complete source code provided with raylib for each of these examples and games to learn other raylib features and techniques.

Implementing Your Own Raylib Games and Simulations

Using what you learn from our code in Exercises 10.19–10.21, you'll enhance our raylib games and simulation and begin creating your own:

- In Exercise 10.22, you'll reimplement your solution to **The Tortoise and the Hare Race** from Exercise 5.54. You'll incorporate the sounds of a traditional horse race, an image of a tortoise and an image of a hare, and you'll play the William Tell Overture in the background during the race.
- In Exercise 10.23, you'll reimplement Section 10.7's high-performance card shuffling and dealing simulation using raylib and attractive public-domain card images to display a deck of cards.
- In Exercises 10.26 and 10.27, you'll attempt enhancements to our SpotOn and Cannon games.
- In Exercises 10.28–10.30, you'll create visualizations for coin tossing, rolling two six-sided dice (producing the sums 2–12) and showing the win/loss results for the hand game Rock, Paper, Scissors, based on the lengths of the games.

Subsequent exercises propose various other games. Get creative—design and build your own games too!

Self-Contained Raylib Windows Environment

Raylib has a self-contained Windows environment with everything you need to create your own games using raylib. The bundle contains:

- the raylib game-programming library,
- the raylib examples and sample games,
- the `gcc` compiler in MinGW[5] (Minimalist GNU for Windows), and
- the Notepad++ text editor, which is preconfigured to enable you to compile and run the raylib example code, raylib sample games and your own games.

You can download the MinGW version of this self-contained environment for free from

```
https://raysan5.itch.io/raylib
```

Compiling and running the raylib examples and sample games in this environment is as simple as opening the C file in Notepad++ and pressing the *F6* key. This displays a window in which you'll see the compilation and execution commands that will run when you click **OK**. For applications that do not have command-line arguments, simply click **OK** to compile and run your code. For applications with command-line arguments, such as our die-rolling simulation, modify the **Execute program** command to place the command-line arguments at the end of the line, then click **OK**.

5. "MinGW (Minimalist GNU for Windows)." Accessed December 16, 2020. `http://www.mingw.org/`.

Installing Raylib on Windows, macOS and Linux

The following URLs contain raylib download and install instructions for Windows, macOS and Linux. Windows users who choose the self-contained environment option do not need to perform these additional install instructions:

- Windows (for those who wish to use raylib with other Windows compilers): `https://github.com/raysan5/raylib/wiki/Working-on-Windows`
- macOS: `https://github.com/raysan5/raylib/wiki/Working-on-macOS`
- Linux: `https://github.com/raysan5/raylib/wiki/Working-on-GNU-Linux`

Raylib Cheatsheet

Though raylib is relatively easy to use, its functions are not extensively documented on `raylib.com`. For a complete list of raylib's functions, see the **raylib cheat sheet**:

`https://www.raylib.com/cheatsheet/cheatsheet.html`

Each function is listed with its prototype followed by a comment that briefly explains its purpose. The cheat sheet also contains the names of raylib's custom types and color constants. You'll notice that raylib's functions are named with a capital first letter. This differs from the C convention of starting function names with a lowercase first letter.

`raylib.h` Header on GitHub

When working with open-source software, occasionally, you may need to look at the source code to get your questions answered. For instance, raylib defines many of its own types—typically as `structs` or `enums`. Most of these are not listed in the cheat-sheet. However, the full raylib source code is available in its GitHub repository:

`https://github.com/raysan5/raylib/`

The header **`raylib.h`** contains the raylib type definitions:

`https://github.com/raysan5/raylib/blob/master/src/raylib.h`

Some of the raylib types you'll use include:

- **`Vector2`**: Contains `x` and `y` members to represent an *x*-*y* coordinate pair.
- **`Rectangle`**: Contains `x`, `y`, `width` and `height` members to represent the upper-left corner, width and height of a rectangle.
- **`Color`**: Colors in raylib are defined using **RGBA colors**. Each color has red (`r`), green (`g`), blue (`b`) and alpha (`a`; transparency) components with values in the range 0–255. See the raylib cheat sheet for a list of raylib's predefined color constants. You may also specify custom colors by creating `Color` objects and setting their `r`, `g`, `b` and `a` members.
- **`Sound`**: Contains members for storing sounds loaded into memory with raylib's `LoadSound` function.
- **`Texture2D`**: Contains members representing a texture loaded into graphics processing unit (GPU) memory.

For these raylib case studies, you do not need to know the `Sound` and Texture2D type details. If you're curious, you can view their definitions in `raylib.h`.

Raylib Uses Frame-By-Frame Animations

In a raylib game, a **game loop** drives a **frame-by-frame animation**. Each loop iteration performs two steps:

1. **Update the game elements for the next animation frame**: In this step, you implement the game logic that determines the game elements' new states. This is where the game-play logic is implemented. Tasks performed here include updating element positions, checking for user-input events (such as mouse clicks), detecting collisions between game elements, updating the score, checking whether the game is over, etc. Element positions are specified as *x*-*y* coordinate pairs within the screen's width and height—*0,0* is the upper-left corner.
2. **Draw the next animation frame's game elements**: In this step, you use raylib's drawing functions to draw the game's elements at their current positions. Raylib stores the pixels of the new animation frame in memory—known as an **off-screen buffer**. When the drawing step completes, raylib displays the off-screen buffer's contents, replacing the previous animation frame on the screen.

Raylib Game Structure

A typical raylib game has the following structure in its `main` function, which we explain below the code listing:

```
1  int main(void) {
2     // initialization
3     InitWindow(screenWidth, screenHeight, "Window Title");
4     InitGame();
5     SetTargetFPS(60);
6
7     // game loop
8     while (!WindowShouldClose()) {
9        UpdateGame(); // update game elements
10       DrawGame(); // draw next animation frame
11    }
12
13    // cleanup
14    UnloadGame(); // release game resources
15    CloseWindow(); // close game window
16 }
```

- The raylib function **`InitWindow`** (line 3) specifies the game window's width in pixels, height in pixels and title.
- A typical raylib sample game contains a user-defined `InitGame` function (line 4). This is where you load sounds, textures and images, initialize the game elements and initialize the variables that maintain the game's state. When a game terminates and the user chooses to play again, you typically call `InitGame` to reset the game state before starting a new game.
- The raylib function **`SetTargetFPS`** (line 5) specifies the number of animation frames raylib tries to draw each second—higher frame rates produce smoother

animations. Today's console games typically try to display 60 frames per second, though some games use more and some fewer. A minimum of 30FPS is recommended for smooth animation.

- The main game loop (lines 8–11) drives the game updates and animation. This loop runs until raylib's `WindowShouldClose` function returns true—when the user closes the window or presses the *Esc* key. This loop updates the game elements, then draws them. Most raylib sample games place the updating code in a function named `UpdateGame` and the drawing code in a function named `DrawGame`. This makes the code easier to maintain.
- When the game loop terminates, `UnloadGame` (line 14) unloads any game resources you loaded in `InitGame`, such as sounds, textures and images.
- Raylib function `CloseWindow` (line 15) releases the game window's resources and closes the game window. Then, the application terminates.

In this code, `InitGame`, `UpdateGame`, `DrawGame` and `UnloadGame` are user-defined functions that define game logic. The raylib code examples and sample games tend to use these names, which follow the capital first letter naming convention used for raylib's functions. We use the same names or similar names in our games and similar names in our simulations that are not games (e.g., `InitSimulation` rather than `InitGame`). We define any other supporting functions with our usual function-naming conventions used throughout this book.

Global Variables and Constants

PERF

For performance, raylib games define the game elements and game state variables as `static` global variables. Such variables are known only from their definitions until the end of the file in which they're defined. Using `static` global variables enables the game's functions to access the game's elements and state directly without passing them to the functions as arguments. This eliminates the overhead of the function call/return mechanisms. As you'll see, even relatively simple games tend to have many game elements and game-state variables. Defining functions with large numbers of parameters tends to make the code harder to maintain, modify and debug.

How to Approach These Case Study Exercises

For each of the first three raylib exercises, we describe what the game does and show screen captures of the game in action. For each game, you should:

1. Read the exercise description to get a sense of the game or simulation.
2. Compile then run the game or simulation several times. For the games, play them to get a feel for how they work.
3. Immerse yourself in the fully coded and commented programs we provide.
4. Tweak the code and rerun it to see the effects of your modifications.

Generally, our code starts with comments that overview the game's functions that we wrote, summarize the raylib functions we use and more.

Interacting with the Raylib Community

Here are some key sites[6] where you can interact with other raylib users and watch raylib videos:

- Discord: `https://discord.gg/VkzNHUE`
- Twitter: `http://www.twitter.com/raysan5`
- Twitch: `http://www.twitch.tv/raysan5`
- Reddit: `https://www.reddit.com/r/raylib`
- Patreon: `https://www.patreon.com/raylib`
- YouTube: `https://www.youtube.com/c/raylib`

Raylib rFXGen Sound-Effect Generator

Raylib has several online tools that help you create items for your games, including icons, textures, graphical user interface elements and layouts and sound effects:

```
https://raylibtech.itch.io/
```

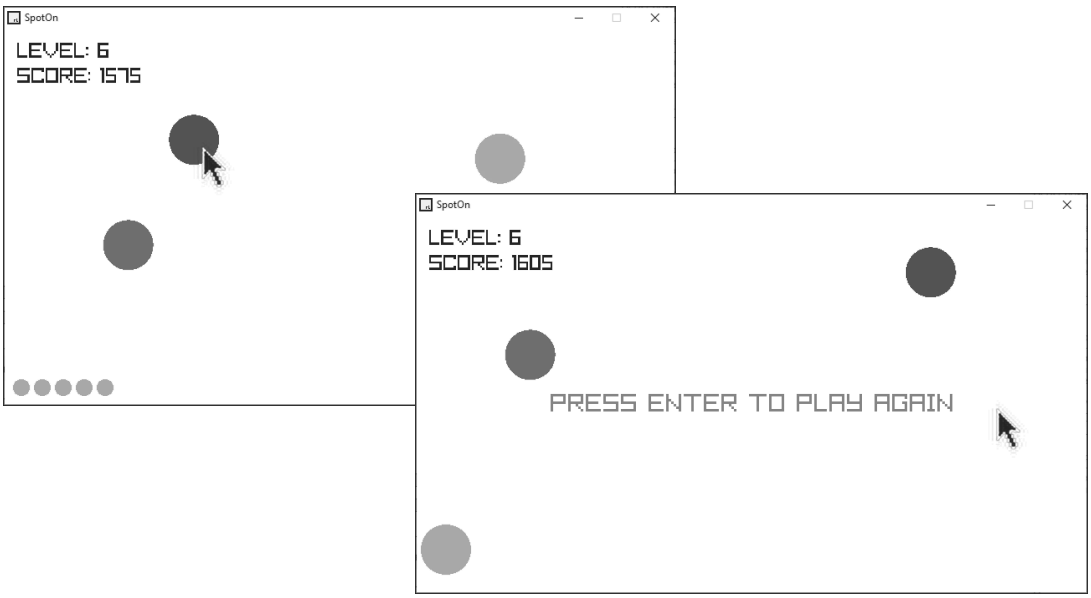
We used **raylib's rFXGen online sound-effect generator**:

```
https://raylibtech.itch.io/rfxgen
```

to create sound effects for our games. You can use the sound effects we provided or create your own.

Game-Programming Case Study Exercise: SpotOn Game

10.19 *(Game Programming Case Study: SpotOn Game)* In this game-programming case study exercise, you'll study our **SpotOn** game, which tests your reflexes by requiring you to click fast-moving spots before they disappear:



6. "README.md." Accessed December 16, 2020. `https://github.com/raysan5/raylib/README.md`.

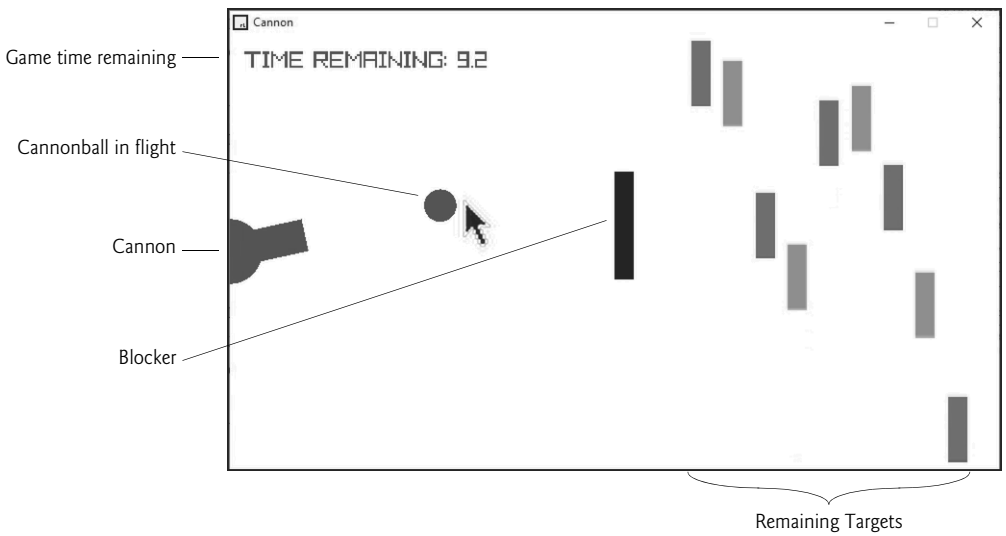
The game begins on level one by displaying three colored spots at random locations. These move at random speeds in random directions. You reach a new level for every 10 spots you click—this increases the spot speed by 5%, making the game increasingly challenging. When you click a spot, the app makes a popping sound, and the spot disappears. You receive points (10 times the current level) for each clicked spot. Accuracy is essential—any click that misses a spot plays a raspberry sound and decreases the score by 15 times the current level. Your current level and score are displayed in the game's top-left corner.

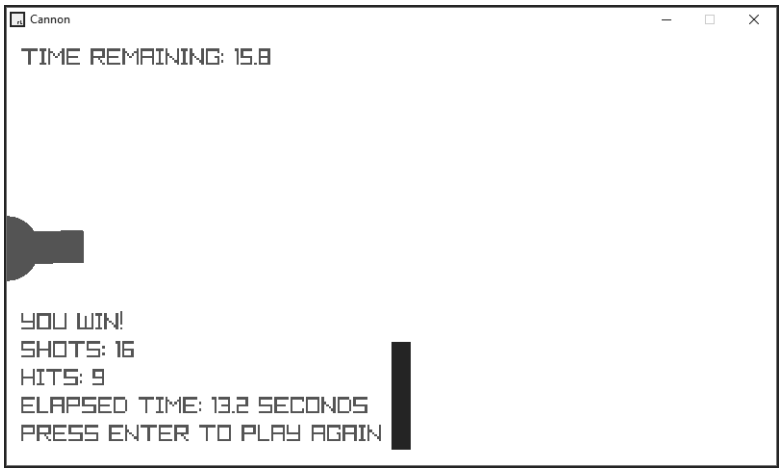
You begin the game with three lives—displayed as small circles in the game's bottom-left corner. If a spot disappears before you click it, you hear a whoosh sound and lose a life. You gain a life for each new level reached, up to a maximum of seven lives. When you lose all your lives, the game ends.

Compile and run the game and play it several times. Next, study this game's code (including extensive comments). Consider tweaking the code to see how your changes affect gameplay. For example, you can change the `spotSpeed` constant's value to make the spots move faster or slower. Finally, improve the game by implementing the enhancements we suggest in Exercise 10.26.

Game-Programming Case Study: Cannon Game

10.20 ***(Game Programming Case Study: Cannon Game)*** In the **Cannon** game, you must destroy nine targets before a ten-second time limit expires:





The game has four types of visual components:

- a **cannon** that you control,
- a **cannonball**,
- **nine targets** that move up and down at various speeds, and
- a **blocker** that moves up and down, defending the targets.

The targets and the blocker move vertically at different but fixed speeds, reversing direction when they hit the screen's top or bottom.

To fire the cannon, you click the mouse. The cannon rotates toward the click point, fires a fast-moving cannonball in a straight line in that direction and plays a **boom sound**. Only one cannonball can be on the screen at a time.

Each time you destroy a target, a **target-destruction sound** plays, the target disappears, and the time remaining increases by a three-second time bonus. The blocker cannot be destroyed. When the cannonball hits the blocker, a **blocker-hit sound** plays, the cannonball bounces back, and the time remaining decreases by a two-second time penalty.

You win by destroying all nine target sections before the time expires. If the timer reaches zero, you lose. At the end of the game, the app displays whether you won or lost and shows the number of shots fired and the elapsed time.

Compile and run the game and play it several times. Next, study this game's code and extensive comments. This application requires some trigonometry to:

- determine the cannon barrel's endpoint, based on its angle, and
- determine the cannonball's x and y increments used to move the cannonball in each animation frame—these also are based on the cannon's barrel angle.

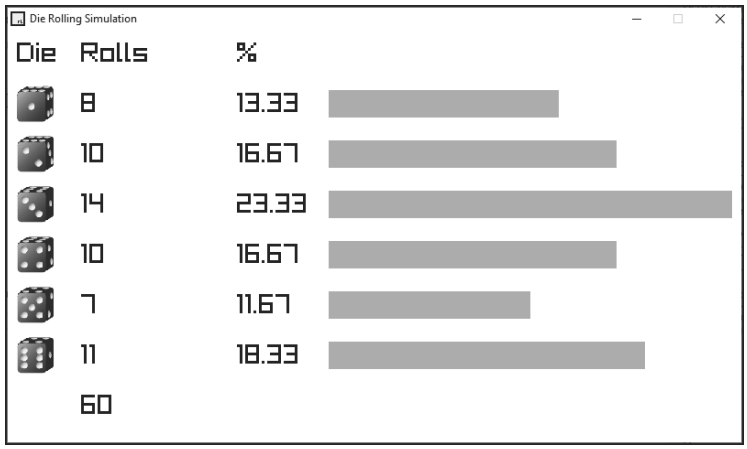
We provide the trigonometry calculations for you.

Consider tweaking the code to see how your changes affect gameplay. For example, you could change how fast the cannonball moves. Finally, improve the game by implementing the enhancements we suggest in Exercise 10.27.

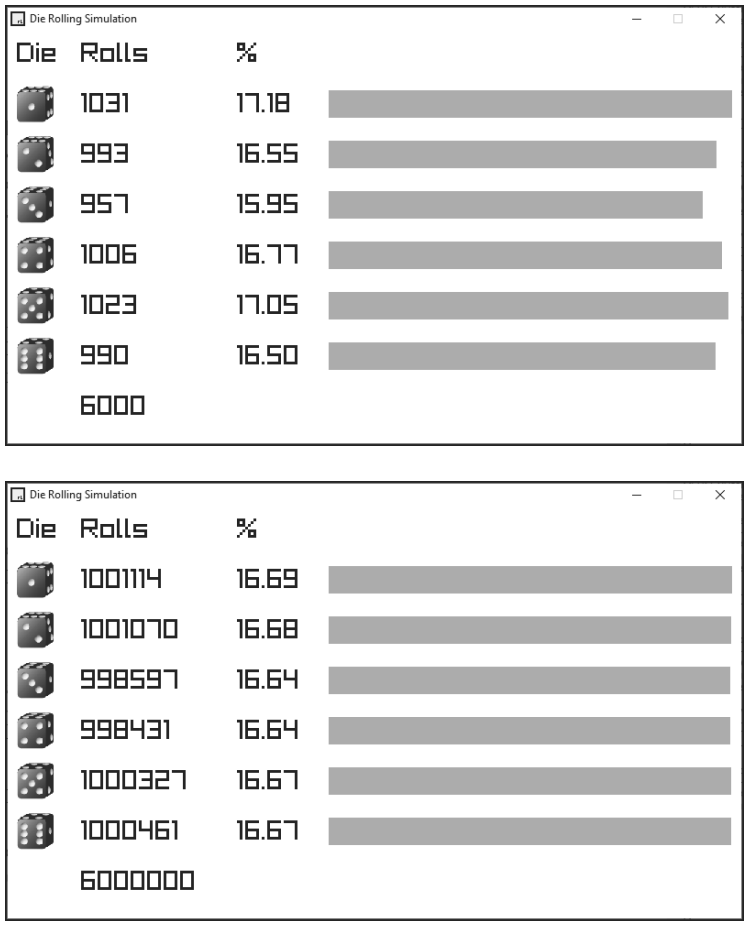
Visualization with raylib—Law of Large Numbers Animation

10.21 ***(Law of Large Numbers Animation)*** In Sections 5.10 and 6.4.7, we used random-number generation to simulate the roll of a six-sided die. In this next raylib case study exercise, you'll use dynamic visualization to make the Law of Large Numbers[7,8] "come alive" in a die-rolling simulation that displays an animated bar chart. As the simulation repeatedly rolls the die, it updates an array of the frequencies with which each face appears. Then, it displays each die face's frequency, its percentage of the total rolls and a bar whose length represents the frequency's magnitude.

For a six-sided die, the face values 1 through 6 each should occur with "equal likelihood"—the probability of each face appearing on any roll is $1/6^{th}$ or approximately 16.67%. If we roll a die 6000 times, we'd expect about 1000 of each face to appear. Like coin tossing, die rolling is random, so faces could occur fewer or more than 1000 times. As the number of die rolls increases, the Law of Large Numbers says that each of the frequencies should approach the expected value of 16.67%. If so, the bars in the bar chart should become nearly identical in length, as shown in the following screen captures of three sample executions for 60, 6000 and 6,000,0000 dice:



7. "Law of large numbers." Accessed December 18, 2020. `https://encyclopediaofmath.org/index.php?title=Law_of_large_numbers`.
8. "Law of large numbers." Accessed December 18, 2020. `https://en.wikipedia.org/wiki/Law_of_large_numbers`.



Running the Simulation on MacOs or Linux

When you execute this simulation, it requires two command-line arguments representing:

- the length of the simulation in animation frames, and
- the number of dice to roll per animation frame.

If the name of the program's executable is `RollDieDynamic`, the following macOS or Linux command will run the simulation for 60 animation frames, rolling one die per frame for a total of 60 rolls:

```
./RollDieDynamic 60 1
```

Similarly, the following will run the simulation for 600 animation frames, rolling 1000 dice per frame for a total of 600,000 rolls:

```
./RollDieDynamic 600 1000
```

Though we do not discuss the details of command-line arguments until Section 15.3, this completely coded simulation provides the statements you need to receive the command-line arguments.

Running the Simulation on MacOs or Linux

For the raylib self-contained Windows environment, perform the following steps to run this simulation:

1. Open `RollDieDynamic.c` in Notepad++.
2. Press the *F6* key.
3. In the **Execute** dialog, modify the last line of the compilation and execution commands to include your command-line arguments, as in:

   ```
   cmd /c IF EXIST $(NAME_PART).exe $(NAME_PART).exe 600 1000
   ```

   Notepad++ replaces `$(NAME_PART)` with the base name of the file you're running—`RollDieDynamic` in this case.
4. Click **OK** to compile and run the program.

Run the Program Several Times

Compile the simulation and run it several times, varying the command-line arguments. Next, study the simulation code (including extensive comments). As in our raylib games, you may pause the simulation at any time by pressing the *P* key and resume the simulation by pressing *P* again. Once you've studied the code, try Exercises 10.28–10.30, where you'll create visualizations for coin tossing, rolling two six-sided dice (producing the sums 2–12) and showing the win/loss results for the hand game Rock, Paper, Scissors, based on the lengths of the games. You may want to use the techniques you've learned to analyze the results of playing popular card games like blackjack and various versions of poker.

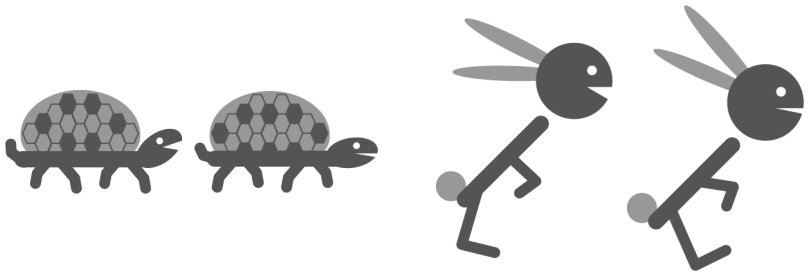
Case Study: The Tortoise and the Hare with raylib—a Multimedia "Extravaganza"

10.22 ***(Multimedia Tortoise and the Hare Race with Raylib)*** In this exercise, you'll use raylib graphics, animation and sound features from Exercises 10.19–10.21 to enhance Exercise 5.54's **The Tortoise and the Hare Race**. You'll incorporate a traditional horse race's sounds and multiple tortoise and hare images to create a fun, animated multimedia "extravaganza." For use in your race, we've provided in this chapter's examples folder a `resources` subfolder containing the following audio clips and images:[9]

- An audio recording we created of the "Call to Post" trumpet piece played at the beginning of a horse race.
- A cannon-firing sound we created for our raylib `Cannon` game. You could use raylib's rFXGen sound generator (`https://raylibtech.itch.io/rfxgen`) to create a firing sound of your own.

9. We created these audios and images. If you prefer, you could search the web for others or create your own. Be sure to comply with the license terms for any media you'll use in your applications.

- An audio clip we created of an announcer saying, "And they're off!" You could record yourself as the announcer saying this and other phrases to play throughout the race, such as "Tortoise pulls ahead!", "Hare pulls ahead!", "Down the stretch they come!", etc.
- A public-domain Wikimedia audio recording of the William Tell Overture[10], which we have edited down to just the gallop portion (bada bum, bada bum, bada bum bum bum…) and placed in this chapter's `resources` folder for you to play during the race.
- Two slightly different tortoise images and two slightly different hare images we created:



We toggle between these images to create simple animations of the animals running. You can see the animations by viewing the `tortoise.gif` and `hare.gif` animated GIF images provided in the `resources` folder with this chapter's examples. Feel free to use these images or have some fun creating your own.

Implementing the Race

Implement the race using the basic raylib game structure you learned in the preceding raylib exercises. In your race, perform the following tasks:

a) Before the race begins, play the trumpet audio of the "Call to Post," signifying that the racers should take their mark. As the "Call to Post" plays, the tortoise and hare should appear from the screen's left side and take their positions.

b) Play the cannon sound to start the race, followed by the announcer saying, "And they're off!" At this point, the race animation begins.

c) Throughout the race, play the provided William Tell Overture's gallop portion in the background repeatedly. See the raylib code sample at `https://www.raylib.com/examples/web/audio/loader.html?name=audio_music_stream` to learn how to play music in the background.

d) As the tortoise and the hare move across the screen, toggle between each animal's two images to make it appear to be running. The tortoise moves slow-

10. "File:Gioachino Rossini, William Tell Overture (military band version, 2000).ogg." Accessed January 2, 2021. `https://commons.wikimedia.org/wiki/File:Gioachino_Rossini,_William_Tell_Overture_(military_band_version,_2000).ogg`.

er than the hare, so you may want to toggle between the tortoise's two images slower than between the hare's two images. When the hare sleeps, stop toggling between its images.

e) When the tortoise and the hare are at the same position, display "OUCH!" for the turtle biting the hare and optionally play a high-pitched "OUCH!" audio clip.

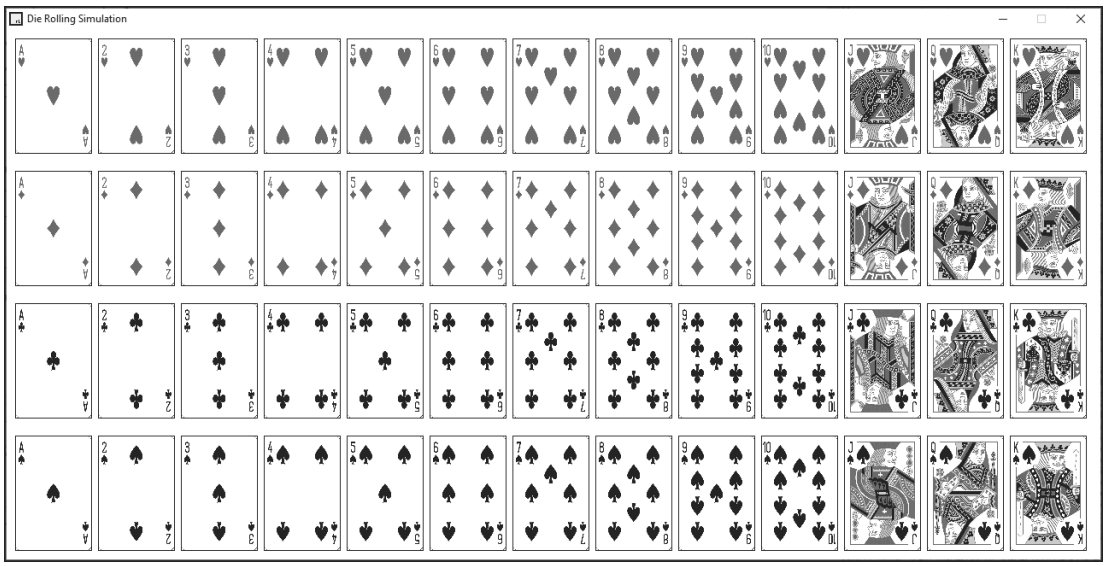
f) If the tortoise wins, display "Tortoise wins!" and optionally play a "Tortoise wins!" audio clip followed by cheering. If the hare wins, display "Hare wins" and optionally play a "Hare wins" audio clip followed by booing. If the race ends in a tie, you may want to favor the tortoise as the underdog or have the announcer say, "It's a tie!"

g) You could play crowd cheering and booing sounds and additional announcer commentary as appropriate throughout the race. You might be able to find public-domain crowd sounds online.

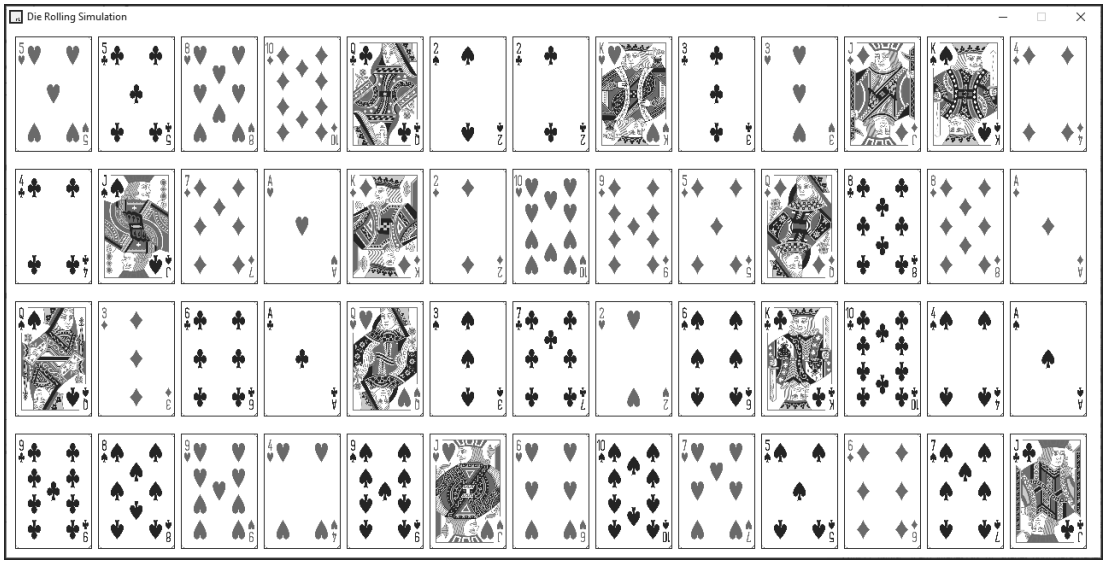
Random-Number Simulation Case Study: High-Performance Card Shuffling and Dealing with Card Images and raylib

10.23 ***(Card Shuffling and Dealing with Card Images and Raylib)*** In Section 10.7, you implemented a high-performance card-shuffling-and-dealing simulation using an array of `Card` objects. In this exercise, you'll incorporate raylib capabilities into your simulation and use them to display attractive, free public-domain card images for each card in the deck. Once you complete this exercise, you'll have the fundamental capabilities you need to begin implementing your favorite card games and to upgrade your solutions to the card-game exercises in earlier chapters.

You'll load the unshuffled 52 card images as raylib `Texture2D` objects, then display them in a 4-by-13 grid:



Each time the user clicks the mouse, shuffle and redisplay the cards:



Public-Domain Card Images from Wikimedia Commons

We downloaded these public-domain[11] card images from:

```
https://commons.wikimedia.org/wiki/
   Category:SVG_English_pattern_playing_cards
```

and provided them for you in the `card_images` subfolder with this chapter's examples. We named each card-image file using the card's face and suit. For example, the images for the Spades suit are named as follows:

- `Ace_of_Spades.png`
- `Deuce_of_Spades.png`
- `3_of_Spades.png`
- `...`
- `Jack_of_Spades.png`
- `Queen_of_Spades.png`
- `King_of_Spades.png`

Implementing the Simulation

Using the basic raylib game framework you learned in Exercises 10.19–10.21, and the image-processing techniques you learned in Exercise 10.21, perform the following tasks:

a) Modify Fig. 10.2's `struct card` definition to include a `Texture2D` member named `image`. This will store raylib's information about a loaded card image.

b) When the application begins executing, initialize the unshuffled deck of cards in your raylib application's `InitSimulation` function. Modify the code that initializes your `deck` array to load each card's image. You can use string-

11. `https://creativecommons.org/publicdomain/zero/1.0/deed.en`.

processing capabilities to assemble each card's *face* and *suit* strings into the card's image file name using the format:

face`_of_`*suit*`.png` (where you fill in the *face* and *suit*)

c) The first time the raylib application's `DrawFrame` function executes, display the unshuffled array of 52 `Card` objects, as shown earlier. You'll need to perform calculations that determine each image's upper-left corner *x*-*y* coordinates.

d) In the raylib application's `Update` function, check whether the user clicked the left mouse button. If so, shuffle the cards. The next call to `DrawFrame` will display the shuffled deck.

Drawing Notes

The following notes will help you implement your simulation:

- For this exercise, set the raylib window's `screenWidth` to `1280` and `screenHeight` to `620` to provide additional room for displaying the card images.
- When drawing each image, set the raylib function `DrawTextureEx`'s `scale` argument to `0.25`. This scales down the images, giving you enough room to draw them in four rows of 13 images each without overlapping one another.
- After drawing each image, use raylib function `DrawRectangleLines` as shown below to place a black border around each image for contrast with the window's white background:

```
DrawRectangleLines(x, y, deck[i].image.width * scale,
   deck[i].image.height * scale, BLACK);
```

Set variable `scale` to `0.25` to draw the rectangle using the same scale as the image.

Additional Raylib Exercises

10.24 ***(Raylib Demos)*** Compile, run and interact with several of raylib's bundled examples located in the `raylib` folder's `examples` subfolder. Study the source code provided for each of those examples to learn more about raylib's features.

10.25 ***(Raylib Sample Games)*** Compile, run and interact with several of raylib's sample games located in the `raylib` folder's `games` subfolder. Study the source code provided for each of those examples to learn more about raylib features. Be creative. Try modifying the games with your own enhancements.

10.26 ***(Project: Enhanced SpotOn Game)*** Try several of the following SpotOn game modifications:

a) Display more spots for higher levels.
b) Use bigger, possibly randomized speed boosts.
c) Give a bonus for destroying multiple spots with one click.
d) Use different point values for each spot color.

e) Make the spots more elusive by allowing them to blink on and off, change direction spontaneously, change size spontaneously and move along nonlinear paths.
f) Intermix smaller, harder-to-click spots.
g) When the user clicks a spot, animate its destruction. For example, it could become concentric circles that fade away from the outside in, or it could become four pizza slices that spread out from the spot's center and fade away.
h) Play a siren sound when the game moves to the next level.
i) Display text for significant events like gaining or losing a life. The text can remain on the screen for a short time, then fade away.
j) Have a specially colored, fast-moving spot. Clicking that spot destroys all the spots on the screen and gives the player a large point bonus.

10.27 ***(Project: Enhanced Cannon Game)*** Try several of the following Cannon game modifications:

a) Display a dashed line showing the cannonball's path.
b) Play a sound when the blocker hits the top or bottom of the screen.
c) Play a sound when a target hits the top or bottom of the screen.
d) Enhance the game to have levels. In each level, increase the number of target pieces.
e) Keep score. Increase the user's score for each target piece hit by 10 times the current level. Decrease the score by 15 times the current level each time the cannonball hits the blocker. Display the score on the screen in the upper-left corner.
f) Add cannonball and target explosion animations each time a cannonball hits a target.
g) Add an explosion animation for the cannonball each time one hits the blocker.
h) When the cannonball hits the blocker, increase the blocker's length by 5%.
i) Make the game more difficult as it progresses by gradually increasing the speed of the targets and the blocker.
j) Increase the number of independently moving blockers between the cannon and the targets.
k) Add a bonus round that lasts for four seconds. Change the color of the targets and add music to indicate that it is a bonus round. If the user hits a target during those four seconds, give the user 1000 bonus points.
l) Allow the user to move the cannon up and down via the arrow keys so it can be fired from different positions.

10.28 ***(Intro to Data Science: Dynamic Visualization of Coin Tossing)*** Modify the Law of Large Numbers die-rolling simulation from Exercise 10.21 to simulate flipping a coin. Use randomly generated 1s and 2s to represent heads and tails, respectively. Run simulations for 20, 200, 20,000 and 2,000,000 coin flips. Do you get approximately 50% heads and 50% tails? Do you see the "Law of Large Numbers" in action?

10.29 ***(Intro to Data Science: Dynamic Visualization of Rolling Two Dice)*** Modify our Law of Large Numbers die-rolling simulation from Exercise 10.21 to simulate rolling two dice. Calculate the sum of the two face values. Each die has a value from 1 to 6, so the sum will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. The following diagram shows the 36 equally likely possible combinations of the two dice and their corresponding sums:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

If you roll the dice 36,000 times:

- The values 2 and 12 each occur with a probability of 1/36th (2.778%), so you should expect about 1000 of each.
- The values 3 and 11 each occur with a probability of 2/36ths (5.556%), so you should expect about 2000 of each, and so on. You should expect about 6000 7s.

Display a dynamic bar plot with bars for each of the sums 2–12 summarizing their frequencies. Run the simulation for 360, 36,000 and 36,000,000 rolls.

10.30 ***(Intro to Data Science Project: Dynamic Visualization of Rock, Paper, Scissors Game Statistics)*** Reimplement your solution to Exercise 5.47 using raylib to create a dynamic bar chart showing the player wins and losses for each of the rounds. Use pairs of green and red bars to indicate player wins and losses, respectively, for each round.

10.31 ***(Project: Brick Game)*** Create a game similar to the cannon game that shoots pellets at a stationary brick wall. The goal is to destroy enough of the wall to shoot the moving target behind it. The faster you break through the wall and hit the target, the higher your score. Include multiple layers to the wall and a small moving target. Keep score. Increase difficulty with each round by building the wall using more layers and smaller bricks and increasing the speed of the moving target.

10.32 ***(Project: Digital Clock)*** Create an app that displays a digital clock on the screen.

10.33 ***(Project: Analog Clock)*** Create an app that displays an analog clock with hour, minute and second hands of appropriate lengths and thicknesses that rotate as the time changes.

This page is intentionally left blank