Documento de lecturas complementarias

La bibliografía relacionada a este material pueden encontrarse en
`https://www.dropbox.com/s/nxp8bva28h9o0f8/prgrmmng_II.pdf?dl=0`

12 de febrero de 2020

Parte I

Structures

10 C Structures, Unions, Bit Manipulation and Enumerations

Objectives

In this chapter, you'll:

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

Outline

10.1 Introduction

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

- `typedef`s—for creating *aliases* for previously defined data types.
- `union`s—similar to structures, but with members that *share* the *same* storage space.
- bitwise operators—for manipulating the bits of integral operands.
- bit fields—`unsigned int` or `int` members of structures or unions for which you specify the number of bits in which the members are stored, helping you pack information tightly.
- enumerations—sets of integer constants represented by identifiers.

10.2 Structure Definitions

Structures are derived data types—they're constructed using objects of other types. Consider the following structure definition:

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

Keyword **struct** introduces a structure definition. The identifier `card` is the **structure tag**, which names the structure definition and is used with `struct` to declare variables of the **structure type**—e.g., `struct card`. Variables declared within the braces of the structure definition are the structure's **members**. Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict (we'll soon see why). Each structure definition *must* end with a semicolon.

Common Programming Error 10.1

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

The definition of `struct card` contains members `face` and `suit`, each of type `char *`. Structure members can be variables of the primitive data types (e.g., `int`, `float`, etc.), or aggregates, such as arrays and other structures. As we saw in Chapter 6, each element of an array must be of the *same* type. Structure members, however, can be of *different* types. For example, the following `struct` contains character array members for an employee's first and last names, an `unsigned int` member for the employee's age, a `char` member that would contain `'M'` or `'F'` for the employee's gender and a `double` member for the employee's hourly salary:

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

10.2.1 Self-Referential Structures

A variable of a `struct` type cannot be declared in the definition of that same `struct` type. A pointer to that `struct` type, however, may be included. For example, in `struct employee2`:

```
struct employee2 {
   char firstName[20];
   char lastName[20];
   unsigned int age;
   char gender;
   double hourlySalary;
   struct employee2 teamLeader; // ERROR
   struct employee2 *teamLeaderPtr; // pointer
};
```

the instance of itself (`teamLeader`) is an error. Because `teamLeaderPtr` is a pointer (to type `struct employee2`), it's permitted in the definition. A structure containing a member that's a pointer to the *same* structure type is referred to as a **self-referential structure**. Self-referential structures are used in Chapter 12, to build linked data structures.

Common Programming Error 10.2

A structure cannot contain an instance of itself.

10.2.2 Defining Variables of Structure Types

Structure definitions do *not* reserve any space in memory; rather, each definition creates a new data type that's used to define variables—like a blueprint of how to build instances of that struct. Structure variables are defined like variables of other types. The definition

```
struct card aCard, deck[52], *cardPtr;
```

declares aCard to be a variable of type struct card, declares deck to be an array with 52 elements of type struct card and declares cardPtr to be a pointer to struct card. After the preceding statement, we've reserved memory for one struct card object named aCard, 52 struct card objects in the deck array and an uninitialized pointer of type struct card. Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding definition could have been incorporated into the struct card definition as follows:

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

10.2.3 Structure Tag Names

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

Good Programming Practice 10.1

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

10.2.4 Operations That Can Be Performed on Structures

The only valid operations that may be performed on structures are:

- assigning struct variables to struct variables of the *same* type (see Section 10.7)—for a pointer member, this copies only the address stored in the pointer.
- taking the address (&) of a struct variable (see Section 10.4).
- accessing the members of a struct variable (see Section 10.4).
- using the sizeof operator to determine the size of a struct variable.

Common Programming Error 10.3

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

Structures may *not* be compared using operators == and !=, because structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are "holes" in a structure, because computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries. A word is a memory unit used to store data in a computer—usually 4 bytes or 8 bytes. Consider the following structure definition, in which sample1 and sample2 of type struct example are declared:

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

A computer with 4-byte words might require that each member of `struct example` be aligned on a word boundary, i.e., at the beginning of a word—this is machine dependent. Figure 10.1 shows a sample storage alignment for a variable of type `struct example` that has been assigned the character `'a'` and the integer `97` (the bit representations of the values are shown). If the members are stored beginning at word boundaries, there's a three-byte hole (bytes 1–3 in the figure) in the storage for variables of type `struct example`. The value in the three-byte hole is *undefined*. Even if the member values of `sample1` and `sample2` are in fact equal, the structures are not necessarily equal, because the undefined three-byte holes are not likely to contain identical values.

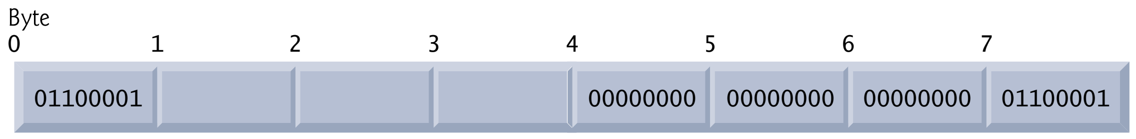

Fig. 10.1 | Possible storage alignment for a variable of type `struct` example showing an undefined area in memory.

Portability Tip 10.1

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

10.3 Initializing Structures

Structures can be initialized using initializer lists as with arrays. To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers. For example, the declaration

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

creates variable `aCard` to be of type `struct card` (as defined in Section 10.2) and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`. If there are *fewer* initializers in the list than members in the structure, the remaining members are automatically initialized to `0` (or `NULL` if the member is a pointer). Structure variables defined outside a function definition (i.e., externally) are initialized to `0` or `NULL` if they're not explicitly initialized in the external definition. Structure variables may also be initialized in assignment statements by assigning a structure variable of the *same* type, or by assigning values to the *individual* members of the structure.

10.4 Accessing Structure Members with . and ->

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

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

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

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

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

Good Programming Practice 10.2

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

Common Programming Error 10.4

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

Common Programming Error 10.5

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

Common Programming Error 10.6

*Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., *cardPtr.suit) is a syntax error. To prevent this problem use the arrow (->) operator instead.*

The program of Fig. 10.2 demonstrates the use of the structure member and structure pointer operators. Using the structure member operator, the members of structure aCard are assigned the values "Ace" and "Spades", respectively (lines 17 and 18). Pointer cardPtr is assigned the address of structure aCard (line 20). Function printf prints the members of structure variable aCard using the structure member operator with variable name aCard, the structure pointer operator with pointer cardPtr and the structure member operator with dereferenced pointer cardPtr (lines 22–24).

```
1  // Fig. 10.2: fig10_02.c
2  // Structure member operator and
3  // structure pointer operator
```

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

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

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

10.12 Anonymous Structures and Unions

Earlier in this chapter we introduced `struct`s and `union`s. C11 now supports anonymous `struct`s and `union`s that can be nested in named `struct`s and `union`s. The members in a nested anonymous `struct` or `union` are considered to be members of the enclosing `struct` or `union` and 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 {
      int nestedMember1;
      int nestedMember2;
   }; // end nested struct
}; // end outer struct
```

For a variable `myStruct` of type `struct MyStruct`, you can access the members as:

```
myStruct.member1;
myStruct.member2;
myStruct.nestedMember1;
myStruct.nestedMember2;
```

10.13 Secure C Programming

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

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, the size of a `struct` variable is *not* necessarily the sum of its members' sizes. Always use `sizeof` to determine the number of bytes in a `struct` variable. As you'll see, we use this technique to manipulate fixed-length records that are written to and read from files in Chapter 11, and to create so-called dynamic data structures in Chapter 12.

- EXP04-C: As we discussed in Section 10.2.4, `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 the condition of a conditional expression (`?:`) 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.

Summary

Section 10.1 Introduction

- **Structures** (p. 437) 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 facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees.

Section 10.2 Structure Definitions

- Keyword **`struct`** introduces a structure definition (p. 438).

Parte II

Exercises

Go to the page 467 (see the next page in the right-hand upper) and solve the exercises: 10.3.

- f) Structures are always passed to functions by reference.
- g) Structures may not be compared by using operators `==` and `!=`.

10.3 Write code to accomplish each of the following:

- a) Define a structure 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` has been defined containing two pointers to type `char`, namely `face` and `suit`. Also, the variable `c` has been defined to be of type `struct card` and the variable `cPtr` has been defined to be of type pointer to `struct card`. Variable `cPtr` has been assigned the address of `c`.

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

- b) Assume that `struct card` has been defined containing two pointers to type `char`, namely `face` and `suit`. Also, the array `hearts[13]` has been defined to be 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 been declared as type `struct person` and variable `c` has been declared as type `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 operator (<<), right-shift operator (>>). n) enumeration.