Summary

Section 10.1 Introduction

- Structures 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.
- The identifier following keyword `struct` is the structure tag, which names the structure definition. The structure tag is used with the keyword `struct` to declare variables of the structure type.
- Variables declared within the braces of the structure definition are the structure's members.
- Members of the same structure type must have unique names.
- Each structure definition must end with a semicolon.
- Structure members can have primitive or aggregates data types.
- A structure cannot contain an instance of itself but may include a pointer to its type.
- 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 to build linked data structures.
- Structure definitions create new data types that are used to define variables.
- Variables of a given structure type can be declared by placing a comma-separated list of variable names between the closing brace of the structure definition and its ending semicolon.
- 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.
- The only valid operations that may be performed on structures are assigning structure variables to variables of the same type, taking the address (`&`) of a structure variable, accessing the members of a structure variable and using the `sizeof` operator to determine the size of a structure variable.

Section 10.3 Initializing Structures

- Structures can be initialized using initializer lists.
- 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).
- Members of structure variables defined outside a function definition are initialized to `0` or `NULL` if they're not explicitly initialized in the external definition.
- Structure variables may 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.

Section 10.4 Accessing Structure Members

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

Section 10.5 Using Structures with Functions

- Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.

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

Section 10.6 `typedef`

- The keyword `typedef` provides a mechanism for creating synonyms for previously defined types.
- Names for structure types are often defined with `typedef` to create shorter type names.
- Often, `typedef` is used to create synonyms for the basic data types. For example, a program requiring 4-byte integers may use type `int` on one system and type `long` on another. Programs designed for portability often use `typedef` to create an alias for 4-byte integers such as `Integer`. The alias `Integer` can be changed once in the program to make the program work on both systems.

Section 10.8 Unions

- A union is declared with keyword `union` in the same format as a structure. Its members share the same storage space.
- The members of a `union` can be of any data type. The number of bytes used to store a `union` must be at least enough to hold the largest member.
- Only one member of a `union` can be referenced at a time. It's your responsibility to ensure that the data in a `union` is referenced with the proper data type.
- The operations that can be performed 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 same type as the first `union` member.

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 are used to manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both signed and unsigned). Unsigned integers are normally used.
- The bitwise operators 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 sets each bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1.
- The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with `0`s; `1`s shifted off the left are lost.
- The right-shift operator 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 sets all `0` bits in its operand to `1` and all `1` bits to `0` in the result.
- 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.

- The symbolic constant `CHAR_BIT` (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 scalable and portable.
- Each binary bitwise operator has a corresponding assignment operator.

Section 10.10 Bit Fields

- C enables you to specify the number of bits in which an `unsigned int` or `int` member of a structure or union is stored. This is referred to as a bit field. Bit fields enable better memory utilization by storing data in the minimum number of bits required.
- 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. 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 to be used as padding in the structure.
- An unnamed bit field with a zero width 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. 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.
- Multiple members of an enumeration can have the same constant value.

Terminology

`.` structure member operator 409
~ bitwise complement operator 423
aggregate 406
arrow operator (`->`) 409
bit field 426
bit field member name 426
bitwise AND (`&`) operator 417
bitwise assignment operator 425
bitwise complement operator (~) 423
bitwise exclusive OR (`^`) operator 417
bitwise inclusive OR (`|`) operator 417
`CHAR_BIT` symbolic constant 420
complement operator (~) 417
derived data type 406
enumeration 631
enumeration constants 429
left-shift operator (`<<`) 417
mask 419
member 407
member name (bit field) 426
one's complement 423
padding 427
pointer to the structure 409
right-shift operator (`>>`) 417
self-referential structure 407
`struct` 407
structure 406
structure member (`.`) operator 409
structure pointer (`->`) operator 409
structure tag 407
structure type 407
`typedef` 411
`union` 415
unnamed bit field 427
unnamed bit field with a zero width 429
width of a bit field 426

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 storage.

c) The bits in the result of an expression using the _____ operator are set to 1 if the corresponding bits in each operand are set to 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 set to 1. Otherwise, the bits are set to zero.
f) Keyword _____ introduces a structure declaration.
g) Keyword _____ is used to create 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 set to 1. Otherwise, the bits are set to zero.
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) A structure member is accessed with either the _____ or the _____ operator.
m) The _____ and _____ operators are used to shift the bits of a value to the left or to the right, respectively.
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) Structures 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 structure is optional.
d) Members of different structures must have unique names.
e) Keyword `typedef` is used to define new data types.
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;
}; // end union values

union values v = { 1.27 };
```

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

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.

10.2 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 the same 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 a)
```
struct part {
   unsigned int partNumber;
   char partName[25];
}; // end struct part
```
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 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 subscript has been omitted. 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) Structure `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 structure called `address` that contains character arrays `streetAddress[ 25 ]`, `city[ 20 ]`, `state[ 3 ]` and `zipCode[ 6 ]`.
d) Structure `student` that contains arrays `firstName[ 15 ]` and `lastName[ 15 ]` and variable `homeAddress` of type `struct address` from part (c).
e) Structure `test` containing 16 bit fields with widths of 1 bit. The names of the bit fields are the letters `a` to `p`.

10.6 Given the following structure 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 can be used to access the structure members in each of the following parts:
a) Member `lastName` of structure `customerRecord`.
b) Member `lastName` of the structure pointed to by `customerPtr`.
c) Member `firstName` of structure `customerRecord`.
d) Member `firstName` of the structure pointed to by `customerPtr`.
e) Member `customerNumber` of structure `customerRecord`.
f) Member `customerNumber` of the structure pointed to by `customerPtr`.
g) Member `phoneNumber` of member `personal` of structure `customerRecord`.
h) Member `phoneNumber` of member `personal` of the structure pointed to by `customerPtr`.
i) Member `address` of member `personal` of structure `customerRecord`.
j) Member `address` of member `personal` of the structure pointed to by `customerPtr`.
k) Member `city` of member `personal` of structure `customerRecord`.
l) Member `city` of member `personal` of the structure pointed to by `customerPtr`.
m) Member `state` of member `personal` of structure `customerRecord`.
n) Member `state` of member `personal` of the structure pointed to by `customerPtr`.