Structures (structs)

Introduction

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

Introduction (Cont.)

- **We’ll also discuss:**
  - **`typedefs`—for creating *aliases* for previously defined data types**

Structure Definitions

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

Structure Definitions (Cont.)

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

Common Programming Error 10.1

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

Structure Definitions (Cont.)

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

Structure Definitions (Cont.)

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

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

Self-Referential Structures

- *A structure cannot contain an instance of itself.*
- For example, a variable of type `struct employee` cannot be declared in the definition for `struct employee`.
- A pointer to `struct employee`, however, may be included.
- For example,
  - ```
    struct employee2 {
       char firstName[ 20 ];
       char lastName[ 20 ];
       unsigned int age;
       char gender;
       double hourlySalary;
       struct employee2 person; // ERROR
       struct employee2 *ePtr; // pointer
    }; // end struct employee2
    ```
- `struct employee2` contains an instance of itself (`person`), which is an error.
    ```

Structure Definitions (Cont.)

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

Defining Variables of Structure Types

- **Structure definitions do *not* reserve any space in memory; rather, each definition creates a new data type that's used to define variables.**
- **Structure variables are defined like variables of other types.**
- **The definition**
    - `struct card aCard, deck[ 52 ], *cardPtr;`

  declares `aCard` to be a variable of type `struct card`, declares `deck` to be an array with 52 elements of type `struct card` and declares `cardPtr` to be a pointer to `struct card`.

Structure Definitions (Cont.)

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

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

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 convenient for declaring new variables of the structure type later in the program.

Operations That Can Be Performed on Structures

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

Common Programming Error 10.2

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

Operations That Can Be Performed on Structures (Cont.)

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

Structure Definitions (Cont.)

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

Structure Definitions (Cont.)

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

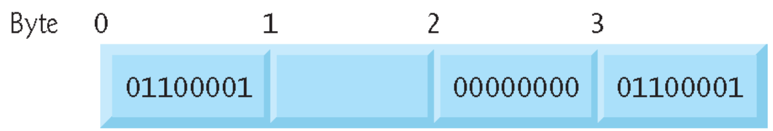

Fig. 10.1 | Possible storage alignment for a variable of type `struct example` showing an undefined area in memory.

Portability Tip 10.1

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

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"`.

Initializing Structures (Cont.)

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

Accessing Structure Members

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

Accessing Structure Members (Cont.)

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

Accessing Structure Members (Cont.)

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

Good Programming Practice 10.2

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

Common Programming Error 10.3

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

Common Programming Error 10.4

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

Common Programming Error 10.5

Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., `*cardPtr.suit`) is a syntax error.

Accessing Structure Members (Cont.)

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

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

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

int main( void )
{
   struct card aCard; // define one struct card variable
   struct card *cardPtr; // define a pointer to a struct card

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

   cardPtr = &aCard; // assign address of aCard to cardPtr

```

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

```
    printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
       cardPtr->face, " of ", cardPtr->suit,
       ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
} // end main
```

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

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

Using Structures with Functions

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

Using Structures with Functions (Cont.)

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

Common Programming Error 10.6

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Performance Tip 10.1

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

typedef

- The keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter type names.
- For example, the statement
    - `typedef struct card Card;`

  defines the new type name `Card` as a synonym for type `struct card`.
- C programmers often use `typedef` to define a structure type, so a structure tag is not required.

typedef (Cont.)

- For example, the following definition

```
typedef struct {
    char *face;
    char *suit;
} Card; // end typedef of Card
```

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

Good Programming Practice 10.3

Capitalize the first letter of `typedef` names to emphasize that they're synonyms for other type names.

typedef (Cont.)

- `Card` can now be used to declare variables of type `struct card`.
- The declaration
  - `Card deck[ 52 ];`

  declares an array of 52 `Card` structures (i.e., variables of type `struct card`).
- Creating a new name with `typedef` does *not* create a new type; `typedef` simply creates a new type name, which may be used as an alias for an existing type name.

typedef (Cont.)

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

Portability Tip 10.2

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

Good Programming Practice 10.4

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