Chapter 10
C Structures, Unions, Bit Manipulation and Enumerations

C How to Program, 8/e

10.2 Structure Definitions

- Structures are derived data types—they are constructed using objects of other types.

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

- Variables declared within the braces of the structure definition are the structure's members.
    ```

10.2 Structure Definitions (Cont.)

- Each structure definition creates a new data type that is used to define variables.

  - `struct employee person, all[ 50 ], *empPtr;`

  - Declare variable with **`struct`** definition

  - ```
    struct employee {
        char firstName[ 20 ];
        char lastName[ 20 ];
        int age;
        char gender;
        double hourlySalary;
    } person, all[50], *empPtr;
    ```
    ```

10.2 Structure Definitions (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

```
struct example {
    char gender;
    int age;
} sample1, sample2;
```

Windows 上中間會有3個bytes沒用到

10.3 Initializing Structures

- ```
  struct example {
      char gender;
      int age;
        char *name;
  };
  ```

- Initialization is similar to array
- `struct example sample1={ ‘M’,25,”John” };`
  ```

10.3 Initializing Structures (Cont.)

- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to `0` (or `NULL` if the member is a pointer).

10.4 Accessing Structure Members

- structure member operator (.)
  - `struct example sample1={ ‘M’,25,”John” };`
  - `printf( "%c", sample1.gender );`
  - `printf( "%d", sample1.age );`
  - `printf( "%s", sample1.name );`
- structure pointer operator (->)
  - `struct example *samplePtr;`
  - `samplePtr = &sample1;`
  - `printf( "%c", samplePtr->gender );`
  - `printf( "%d", samplePtr->age );`
  - `printf( "%s", samplePtr->name );`

10.4 Accessing Structure Members

- A new copy of a structure's value can be made by simply assigning one structure to another
- Example
  - `struct example sample1={ 'M',25,"John" };`
  - `struct example sample2;`
  - `sample2=sample1;`

10.5 Using Structures with Functions

- Structures as input parameters
  - pass by value

```
void print_struct(struct example  sample)
{
        printf(“name=%s\n”, sample.name);
        printf(“age=%d\n”, sample.age);
        printf(“gener=%c\n”, sample.gener);
}
```

10.5 Using Structures with Functions

- Structures as input parameters
  - pass by reference

```
int scan_struct(struct example *sPtr)
{
    int result;
    result = scanf(“%c%d%s”, &sPtr->gender,
                              &sPtr->age,
                              sPtr->name);
    return(result);
}
```

10.5 Using Structures with Functions

- The equality(==) and inequality(!=) operators cannot be applied to a structured type as a unit

```
int cmp_struct(struct example sample1, struct example sample2)
{
    if(sample1.gender == sample2.gender &&
        sample1.age == sample2.age &&
        strcmp(sample1.name, sample2.name)==0)
        return 1;
    else
        return 0;
}
```

10.6 typedef

- The keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- For example, the statement
  - **`typedef struct card Card;`**
- Card can be used to declare variables
  - `Card oneCard;`
  - `Card deck[ 52 ];`

10.6 typedef (Cont.)

- The following definition
  - ```
    typedef struct {
       char *face;
       char *suit;
    } Card;
    ```

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

10.8 Unions

- A union is a derived data type—like a structure—with members that share the same storage space.
  - ```
    union number {
        int x;
        double y;
    };
    ```
- In a declaration, a union may be initialized with a value of the same type as the first union member.
  - `union number value = { 10 };`
    ```

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

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

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

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

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

```c

    value.y = 100.0; // put a double into the same union
    printf("%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
       "Put 100.0 in the floating member",
       "and print both members.",
       "int:", value.x,
       "double:", value.y);
}
```

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

double:
  -92559592117433136000000000000000000000000000000000000000000000.000000


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

double:
  100.000000
```

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

10.9 Bitwise Operators

- The bitwise operators are
  - bitwise AND (&)
  - bitwise inclusive OR (|)
  - bitwise exclusive OR (^)
  - left shift (<<)
  - right shift (>>)
  - complement (~)

<table>
<tr><th colspan="2">Operator</th><th>Description</th></tr>
<tr><td>&</td><td>bitwise AND</td><td>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.</td></tr>
<tr><td>|</td><td>bitwise inclusive OR</td><td>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.</td></tr>
<tr><td>^</td><td>bitwise exclusive OR (also known as bitwise XOR)</td><td>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.</td></tr>
<tr><td><<</td><td>left shift</td><td>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.</td></tr>
<tr><td>>></td><td>right shift</td><td>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.</td></tr>
<tr><td>~</td><td>complement</td><td>All 0 bits are set to 1 and all 1 bits are set to 0.</td></tr>
</table>

Fig. 10.6 | Bitwise operators.

10.9 Bitwise Operators (Cont.)

- The program of Fig. 10.7 prints an `unsigned` integer in its binary representation in groups of eight bits each.
- The results of bitwise operations are machine dependent.

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

void displayBits(unsigned int value); // prototype

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

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

   displayBits(x);
}

```

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

```c
// display bits of an unsigned int value
void displayBits(unsigned int value)
{
   // define displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;

   printf("%10u = ", value);

   // loop through bits
   for (unsigned int c = 1; c <= 32; ++c) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1; // shift value left by 1

      if (c % 8 == 0) { // output space after 8 bits
         putchar(' ');
      }
   }

   putchar('\n');
}
```

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

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

10.9 Bitwise Operators (Cont.)

- 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.

Fig. 10.14 | The bitwise assignment operators.

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

Fig. 10.15 | Operator precedence and associativity.

10.10 Bit Fields

- Bit field: C enables you to specify the number of bits in which an `unsigned` or `int` member of a structure or union is stored.
  - Bit field members *must* be declared as `int` or `unsigned`.
    - ```c
      struct bitCard {
          unsigned face : 4;
          unsigned suit : 2;
          unsigned color : 1;
      };
      ```
      ```

10.10 Bit Fields (Cont.)

- An unnamed bit field with a zero width is used to align the next bit field on a new storage-unit boundary.

  - ```
    struct example {
       unsigned int a : 13;
       unsigned int   : 0;
       unsigned int b : 4;
    };
    ```
    ```

10.10 Enumeration Constants

- Enumeration is a user-defined type.
- Values in an `enum` start with `0`, unless specified otherwise, and are incremented by `1`.

- For example, the enumeration
  - ```
    enum months {
        JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
        OCT, NOV, DEC };
    ```

  creates a new type, `enum months`, in which the identifiers are set to the integers `0` to `11`, respectively.
    ```

10.11 Enumeration Constants (Cont.)

- To number the months 1 to 12, use the following enumeration:
  - ```
    enum months {
        JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
        SEP, OCT, NOV, DEC };
    ```
- Since the first value in the preceding enumeration is explicitly set to `1`, the remaining values are incremented from `1`, resulting in the values `1` through `12`.
- The identifiers in an enumeration must be unique.
    ```

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

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

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

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

Fig. 10.18 | Using an enumeration. (Part 1 of 2.)

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

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