Chapter 10 - Structures, Unions, Bit Manipulations, and Enumerations

Outline

10.1 Introduction

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees

10.2 Structure Definitions

- Example

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

  - **`struct`** introduces the definition for structure card
  - **`card`** is the structure name and is used to declare variables of the structure type
  - **`card`** contains two members of type **`char *`**
    - These members are **`face`** and **`suit`**

10.2 Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to declare structure variables
- Declarations
  - Declared like other variables:

```
card oneCard, deck[ 52 ], *cPtr;
```

  - Can use a comma separated list:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

10.2 Structure Definitions

- Valid Operations
  - Assigning a structure to a structure of the same type
  - Taking the address (**`&`**) of a structure
  - Accessing the members of a structure
  - Using the **`sizeof`** operator to determine the size of a structure

10.3 Initializing Structures

- Initializer lists
  - Example:

```
card oneCard = { "Three", "Hearts" };
```

- Assignment statements
  - Example:

```
card threeHearts = oneCard;
```

  - Could also declare and initialize **threeHearts** as follows:

```
card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

10.4 Accessing Members of Structures

- Accessing structure members
  - Dot operator (**.**) used with structure variables

    ```
    card myCard;
    printf( "%s", myCard.suit );
    ```

  - Arrow operator (**->**) used with pointers to structure variables

    ```
    card *myCardPtr = &myCard;
    printf( "%s", myCardPtr->suit );
    ```

  - **myCardPtr->suit** is equivalent to

    ```
    ( *myCardPtr ).suit
    ```

10.5 Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

10.6 typedef

- **typedef**
  - Creates synonyms (aliases) for previously defined data types
  - Use **typedef** to create shorter type names
  - Example:

```
typedef struct Card *CardPtr;
```

  - Defines a new type name **CardPtr** as a synonym for type **struct Card ***
  - **typedef** does not create a new data type
    - Only creates an alias

10.7 Example: High-Performance Card-shuffling and Dealing Simulation

- Pseudocode:
  - Create an array of card structures
  - Put cards in the deck
  - Shuffle the deck
  - Deal the cards

```c
/* Fig. 10.3: fig10_03.c
   The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

typedef struct card Card;

void fillDeck( Card * const, const char *[],
               const char *[] );
void shuffle( Card * const );
void deal( const Card * const );

int main()
{
   Card deck[ 52 ];
   const char *face[] = { "Ace", "Deuce", "Three",
                          "Four", "Five",
                          "Six", "Seven", "Eight",
                          "Nine", "Ten",
                          "Jack", "Queen", "King"};
   const char *suit[] = { "Hearts", "Diamonds",
                          "Clubs", "Spades"};

   srand( time( NULL ) );
```

Outline

1. Load headers

1.1 Define `struct`

1.2 Function prototypes

1.3 Initialize `deck[]` and `face[]`

1.4 Initialize `suit[]`

Outline

2. fillDeck

2.1 shuffle

definitions

```
   fillDeck( deck, face, suit );
   shuffle( deck );
   deal( deck );
   return 0;
}

void fillDeck( Card * const wDeck, const char * wFace[],
               const char * wSuit[] )
{
   int i;

   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   }
}

void shuffle( Card * const wDeck )
{
   int i, j;
   Card temp;

   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wDeck[ i ];
      wDeck[ i ] = wDeck[ j ];
      wDeck[ j ] = temp;
   }
}
```

Put all 52 cards in the deck. **face** and **suit** determined by remainder (modulus).

Select random number between **0** and **51**. Swap element **i** with that element.

Outline

```c

void deal( const Card * const wDeck )
{
   int i;

   for ( i = 0; i <= 51; i++ )
      printf( "%5s of %-8s%c", wDeck[ i ].face,
              wDeck[ i ].suit,
              ( i + 1 ) % 2 ? '\t' : '\n' );
}
```

Cycle through array and print out data.

Outline

Program Output

```
Eight of Diamonds        Ace of Hearts
Eight of Clubs          Five of Spades
Seven of Hearts        Deuce of Diamonds
  Ace of Clubs           Ten of Diamonds
Deuce of Spades          Six of Diamonds
Seven of Spades        Deuce of Clubs
 Jack of Clubs           Ten of Spades
 King of Hearts         Jack of Diamonds
Three of Hearts        Three of Diamonds
Three of Clubs          Nine of Clubs
  Ten of Hearts        Deuce of Hearts
  Ten of Clubs         Seven of Diamonds
  Six of Clubs         Queen of Spades
  Six of Hearts        Three of Spades
 Nine of Diamonds        Ace of Diamonds
 Jack of Spades         Five of Clubs
 King of Diamonds      Seven of Clubs
 Nine of Spades         Four of Hearts
  Six of Spades        Eight of Spades
Queen of Diamonds       Five of Diamonds
  Ace of Spades         Nine of Hearts
 King of Clubs          Five of Hearts
 King of Spades         Four of Diamonds
Queen of Hearts        Eight of Hearts
 Four of Spades         Jack of Hearts
 Four of Clubs         Queen of Clubs
```

10.8 Unions

- **`union`**
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a **`union`** share space
  - Conserves storage
  - Only the last data member defined can be accessed
- **`union`** declarations
  - Same as struct

```
union Number {
   int x;
   float y;
};
union Number value;
```

10.8 Unions

- Valid **`union`** operations
  - Assignment to **`union`** of same type: **`=`**
  - Taking address: **`&`**
  - Accessing union members: **`.`**
  - Accessing members using pointers: **`->`**

Outline

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

union number {
   int x;
   double y;
};

int main()
{
   union number value;

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

   value.y = 100.0;
   printf( "%s\n%s\n%s%d\n%s%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int:    ", value.x,
           "double:\n", value.y );
   return 0;
}
```

1. Define union

1.1 Initialize variables

2. Set variables

3. Print

Outline

Program Output

```
Put a value in the integer member
and print both members.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000.00000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

10.9 Bitwise Operators

- All data represented internally as sequences of bits
  - Each bit can be either `0` or `1`
  - Sequence of 8 bits forms a byte

Operator	Name	Description
`&`	bitwise AND	The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`.
`|`	bitwise OR	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	The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from 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.
`~`	One's complement	All `0` bits are set to `1` and all `1` bits are set to `0`.

```c
/* Fig. 10.9: fig10_09.c
   Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR and bitwise complement operators */
#include <stdio.h>

void displayBits( unsigned );

int main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
   printf( "The result of combining the following\n" );
   displayBits( number1 );
   displayBits( mask );
   printf( "using the bitwise AND operator & is\n" );
   displayBits( number1 & mask );

   number1 = 15;
   setBits = 241;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator | is\n" );
   displayBits( number1 | setBits );

   number1 = 139;
   number2 = 199;
   printf( "\nThe result of combining the following\n" );
```


1. Function prototype

1.1 Initialize variables

2. Function calls

2.1 Print

2.1 Print

3. Function definition

```
   displayBits( number1 );
   displayBits( number2 );
   printf( "using the bitwise exclusive OR operator ^ is\n" );
   displayBits( number1 ^ number2 );

   number1 = 21845;
   printf( "\nThe one's complement of\n" );
   displayBits( number1 );
   printf( "is\n" );
   displayBits( ~number1 );

   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 31;

   printf( "%7u = ", value );

   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         putchar( ' ' );
   }

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

MASK created with only one set bit

i.e. (**10000000 00000000**)

The **MASK** is constantly **AND**ed with **value**.

MASK only contains one bit, so if the **AND** returns true it means **value** must have that bit.

value is then shifted to test the next bit.

Program Output

```
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
```

10.10 Bit Fields

- Bit field
  - Member of a structure whose size (in bits) has been specified
  - Enable better memory utilization
  - Must be declared as **`int`** or **`unsigned`**
  - Cannot access individual bits
- Declaring bit fields
  - Follow **`unsigned`** or **`int`** member with a colon (**`:`**) and an integer constant representing the width of the field
  - Example:

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

10.10 Bit Fields

- Unnamed bit field
  - Field used as padding in the structure
  - Nothing may be stored in the bits

```
struct Example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
}
```

  - Unnamed bit field with zero width aligns next bit field to a new storage unit boundary

10.11 Enumeration Constants

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at **0** and are incremented by **1**
    - Values can be set explicitly with **=**
    - Need unique constant names
  - Example:

```
enum Months { JAN = 1, 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 **1 to 12**
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)

Outline

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

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

int main()
{
   enum months month;
   const char *monthName[] = { "", "January", "February",
                               "March", "April", "May",
                               "June", "July", "August",
                               "September", "October",
                               "November", "December" };

   for ( month = JAN; month <= DEC; month++ )
      printf( "%2d%11s\n", month, monthName[ month ] );

   return 0;
}
```

1. Define enumeration

1.1 Initialize variable

2. Loop

2.1 Print

Program Output

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