Structures, Unions, Bit Manipulations, and Enumerations

Introduction

- In C, we can create our own data types
- If programmers do a good job of this, the end user does not even have to know what is in the data type
- **struct** is used to describe a new data type.
- **typedef** is used to associate a name with it.
- **int**, **double** and **char** are types of variables. With **struct** you can create your own. It is a new way to extend the C programming language.

Structure

- 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

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`**

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

Structure Declaration

- 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;
```


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
```


Example 8A

```
/* 8A
Using the structure member and
   structure pointer operators */
#include <stdio.h>
#include <conio.h>
struct card {
   char *face;
   char *suit;
};
int main()
{  struct card a;
   struct card *aPtr;
   a.face = "Ace";
   a.suit = "Spades";
   aPtr  = &a;
   printf( "%s%s%s\n%s%s%s\n%s%s%s\n",
         a.face, " of ", a.suit,
         aPtr->face, " of ", aPtr->suit,
         ( *aPtr ).face, " of ", ( *aPtr ).suit );
   getch();
   return 0;
}
```

Output Result

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

typedef

- typedef allows us to associate a name with a structure (or other data type).
- 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

Example

```
typedef struct line {
   int x1, y1;
   int x2, y2;
} LINE;

int main()
{
  LINE line1;
}
```

line1 is now a structure of line type
This is what was happening with all that FILE * stuff

Example 8B

```
/*8B
 The card shuffling and dealing program using structures */
#include<stdio.h>
#include<stdlib.h>
#include <time.h>
#include<conio.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 );
```


Example 8B (cont')

```
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 ) );

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


Example 8B (cont')

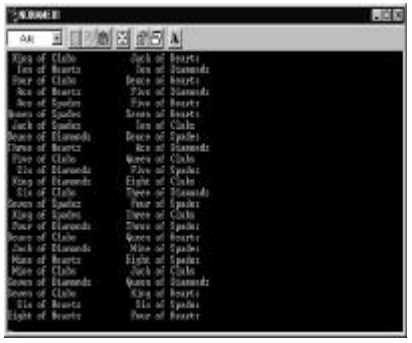
```
void fillDeck ( Card * const wDeck, const char * wFace[],
                const char * wSuit[] )
{   int i;
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   }
}
void shuffle( Card * const wDeck )
{
   int i, j;
   Card temp;
   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wDeck [ i ];
      wDeck[ i ] = wDeck [ j ];
      wDeck[ j ] = temp;
   }
}
```


Example 8B (cont')

```
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' );
}
```

Output Result



Unions

- 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

- **`union`** declarations
  - Same as struct

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


Valid union Operations

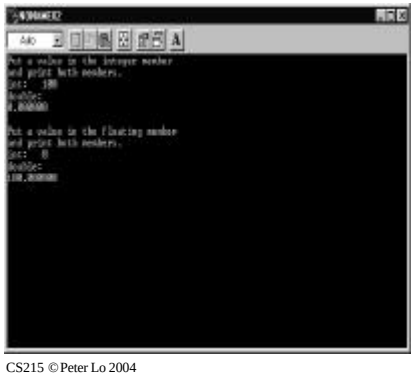
- Valid **`union`** operations
  - Assignment to **`union`** of same type: **`=`**
  - Taking address: **`&`**
  - Accessing union members: **`.`**
  - Accessing members using pointers: **`->`**

Example 8C

```
/* 8 C: An example of a union */
#include<stdio .h>
#include<conio .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 );
   getch ();
   return 0;
}
```


Output Result



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

x	y	AND x&y	OR x|y	XOR x^y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

x	Complement ~x
0	1
1	0

Bitwise Operators

Operator	Name	Description
&	twise AND	he bits in the result are set to **1** if the corresponding bits in the o operands are both **1**.
|	twise OR	he bits in the result are set to **1** if at least one of the rresponding bits in the two operands is **1**.
^	twise exclusive OR	he bits in the result are set to **1** if exactly one of the rresponding bits in the two operands is **1**.
<<	ft shift	ifts the bits of the first operand left by the number of bits ecified by the second operand; fill from right with **0** bits.
>>	ght shift	ifts the bits of the first operand right by the number of bits ecified by the second operand; the method of filling from the ft is machine dependent.
~	ne's complement	ll **0** bits are set to **1** and all **1**bits are set to **0**.

Example 8D

```
/* 8D: Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR and bitwise complement operators */
#include <stdio.h>
#include <conio.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 &\n" );
   displayBits( number1 & mask );
  number1 = 21845;
   printf( "\nThe one's complement of\n" );
```


Example 8D (cont')

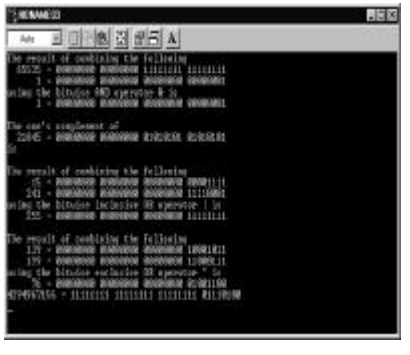
```
   displayBits( number1 );
   printf( "is\n" );
   number1 = 15;
   setBits= 241;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator |\n" );
   displayBits( number1 |setBits);

   number1 = 139;
   number2 = 199;
   printf( "\nThe result of combining the following\n" );
  displayBits( number1 );
   displayBits( number2 );
   printf( "using the bitwise exclusive OR operator ^\n" );
   displayBits( number1 ^ number2 );
  displayBits( ~number1 );
   getch();
   return 0;
}
```


Example 8D (cont')

```
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' );
}
```


Output Result



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

Enumeration Constants

- 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)

Example 8F

```
/* 8F
Using an enumeration type */
#include <stdio.h>
#include <conio.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 ] );
  getch();
  return 0;
}
```


Output Result