Chapter 7 - Pointers

Outline

7.1 Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

7.2 Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)

  - Pointers contain *address* of a variable that has a specific value (indirect reference)

  - Indirection - referencing a pointer value

7.2 Pointer Variable Declarations and Initialization (II)

- Pointer declarations
  - **`*`** used with pointer variables

    ```
    int *myPtr;
    ```

  - Declares a pointer to an **`int`** (pointer of type **`int *`**)
  - Multiple pointers, multiple **`*`**

    ```
    int *myPtr1, *myPtr2;
    ```

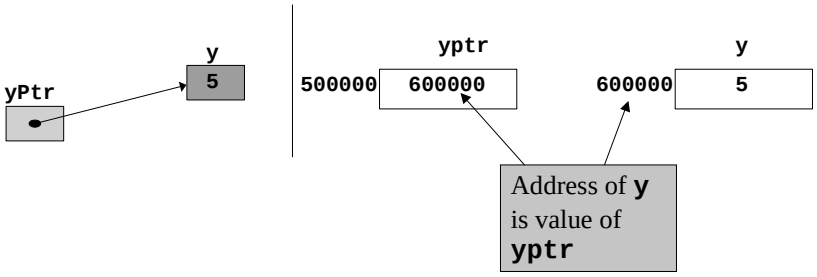
  - Can declare pointers to any data type
  - Initialize pointers to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** - points to nothing (**`NULL`** preferred)

7.3 Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;  //yPtr gets address of y
```

  - **yPtr** "points to" **y**



7.3 Pointer Operators (II)

- ***** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand *points* to
    ***yptr** returns **y** (because **yptr** points to **y**)
  - ***** can be used for assignment
    - Returns alias to an object

```
*yptr = 7; // changes y to 7
```

  - Dereferenced pointer (operand of *****) must be an *lvalue* (no constants)

7.3 Pointer Operators (III)

- ***** and **&** are inverses
  - They cancel each other out

```
*&yptr  ->  * (&yptr) -> * (address of yptr)->
  returns alias of what operand points to -> yptr

&*yptr -> &(*yptr) -> &(y) -> returns address of y,
  which is yptr -> yptr
```


Outline

```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;        /* a is an integer */
8     int *aPtr;    /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;    /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are inverses of "
20            "each other.\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0;
24 }
```

The address of **a** is the value of **aPtr.**

1. Declare variables

2 Initialize variables

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

Program Output

7.4 Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer

- ***** operator
  - Used as alias/nickname for variable inside of function

```
void double(int *number)
 {
  *number = 2 * (*number);
 }
```

***number** used as nickname for the variable passed

```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference
3     with a pointer argument */
4
5  #include <stdio.h>
6
7  void cubeByReference( int * );   /* p
8
9  int main()
10 {
11    int number = 5;
12
13    printf( "The original value of number is %d", number );
14    cubeByReference( &number );
15    printf( "\nThe new value of number is %d\n", number );
16
17    return 0;
18 }
19
20 void cubeByReference( int *nPtr )
21 {
22    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
23 }
```

Notice how the address of **number** is given - **cubeByReference** expects a pointer (an address of a variable).

Inside **cubeByReference**, ***nPtr** is used (***nPtr** is **number**).

Outline

1. Function prototype - takes a pointer to an **int**.

1.1 Initialize variables

2. Call function

3. Define function

Program Output

```
The original value of number is 5
The new value of number is 125
```

7.5 Using the Const Qualifier with Pointers

- **const** qualifier - variable cannot be changed
  - Good idea to have **const** if function does not need to change a variable
  - Attempting to change a **const** is a compiler error
- **const** pointers - point to same memory location
  - Must be initialized when declared

```
int *const myPtr = &x;
```

    - Type **int *const -** constant pointer to an **int**

```
const int *myPtr = &x;
```

    - Regular pointer to a **const int**

```
const int *const Ptr = &x;
```

    - **const** pointer to a **const int**
    - **x** can be changed, but not ***Ptr**

```
1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to
3     non-constant data */
4
5  #include <stdio.h>
6
7  int main()
8  {
9     int x, y;
10
11    int * const ptr = &x; /* ptr is a constant pointer to an
12                             integer. An integer can be modified
13                             through ptr, but ptr always points
14                             to the same memory location. */
15    *ptr = 7;
16    ptr = &y;
17
18    return 0;
19 }
```

Changing ***ptr** is allowed - **x** is not a constant.

Changing **ptr** is an error - **ptr** is a constant pointer.

```
FIG07_13.c:
Error E2024 FIG07_13.c 16: Cannot modify a const object in
function main
*** 1 errors in Compile ***
```

Outline

1. Declare variables

1.1 Declare const pointer to an int.

2. Change *ptr (which is x).

2.1 Attempt to change ptr.

3. Output

Program Output

7.6 Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - Swap two elements
  - **`swap`** function must receive address (using **&**) of array elements
    - Array elements have call-by-value default
  - Using pointers and the **`*`** operator, **`swap`** can switch array elements

- Psuedocode

  Initialize array

  print data in original order

  Call function bubblesort

  print sorted array

  Define bubblesort

7.6 Bubble Sort Using Call-by-reference (II)

- **`sizeof`**
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if **`sizeof(int) = 4 bytes, then`**

```
int myArray[10];
printf( "%d", sizeof( myArray ) );
```

    will print 40

- **`sizeof`** can be used with
  - Variable names
  - Type name
  - Constant values

```
1  /* Fig. 7.15: fig07_15.c
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <stdio.h>
5  #define SIZE 10
6  void bubbleSort( int *, const int );
7
8  int main()
9  {
10
11    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
12    int i;
13
14    printf( "Data items in original order\n" );
15
16    for ( i = 0; i < SIZE; i++ )
17       printf( "%4d", a[ i ] );
18
19    bubbleSort( a, SIZE );          /* sort the array */
20    printf( "\nData items in ascending order\n" );
21
22    for ( i = 0; i < SIZE; i++ )
23       printf( "%4d", a[ i ] );
24
25    printf( "\n" );
26
27    return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32    void swap( int *, int * );
```

Bubblesort gets passed the address of array elements (pointers). The name of an array is a pointer.

Outline

1. Initialize array

1.1 Declare variables

2. Print array

2.1 Call bubbleSort

2.2 Print array

```
33    int pass, j;
34    for ( pass = 0; pass < size - 1; pass++ )
35
36       for ( j = 0; j < size - 1; j++ )
37
38          if ( array[ j ] > array[ j + 1 ] )
39             swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44    int hold = *element1Ptr;
45    *element1Ptr = *element2Ptr;
46    *element2Ptr = hold;
47 }
```

Outline

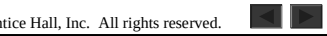
3. Function definitions

Program Output

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45
```

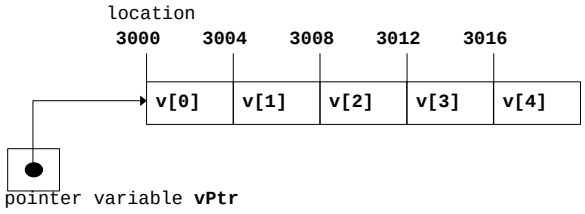
7.7 Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (**++** or **--**)
  - Add an integer to a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array



7.7 Pointer Expressions and Pointer Arithmetic (II)

- 5 element **int** array on machine with 4 byte **int**s
  - **vPtr** points to first element **v[0]**
    at location **3000**. (**vPtr = 3000**)
  - **vPtr +=2;** sets **vPtr** to **3008**
    - **vPtr** points to **v[2]** (incremented by 2), but machine has 4 byte **int**s.



7.7 Pointer Expressions and Pointer Arithmetic (III)

- Subtracting pointers
  - Returns number of elements from one to the other.

```
vPtr2 = v[2];
vPtr = v[0];
vPtr2 - vPtr == 2.
```

- Pointer comparison (**<**, **==** , **>**)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to **0**

7.7 Pointer Expressions and Pointer Arithmetic (IV)

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to **void** (type **void ***)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to **void** pointer
    - **void** pointers cannot be dereferenced

7.8 The Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations

- Declare an array **b[5]** and a pointer **bPtr**

  bPtr = b;

  Array name is actually an address - i.e. address of the first element

  OR

  bPtr = &b[0]

  Explicitly assign **bPtr** to address of first element

7.8 The Relationship Between Pointers and Arrays (II)

- Element **b[n]**
  - can be accessed by ***(bPtr + n)**
  - **n** - offset (pointer/offset notation)
  - Array itself can use pointer arithmetic.

    b[3] same as ***(b + 3)**
  - Pointers can be subscripted (pointer/subscript notation)

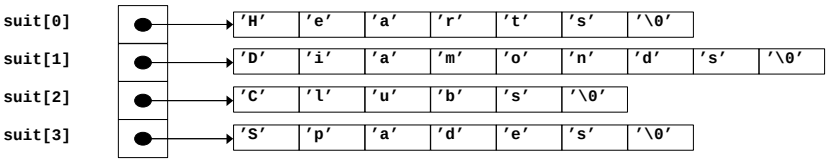
    bPtr[3] same as **b[3]**

7.9 Arrays of Pointers

- Arrays can contain pointers - array of strings

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades" };
```

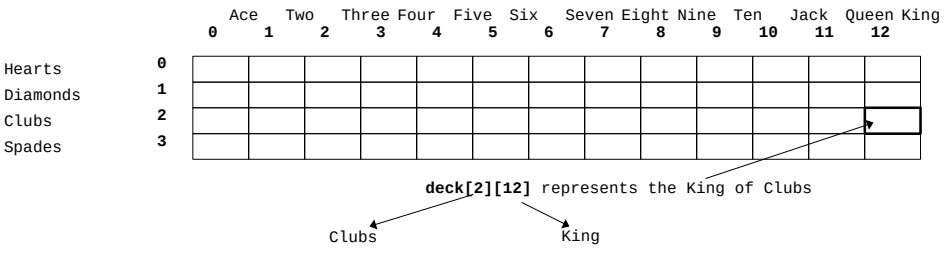
  - String: pointer to first character
  - **`char *`** - each element of **`suit`** is a pointer to a **`char`**
  - Strings not actually in array - only *pointers* to string in array



- **`suit`** array has a fixed size, but strings can be of any size.

7.10 Case Study: A Card Shuffling and Dealing Simulation

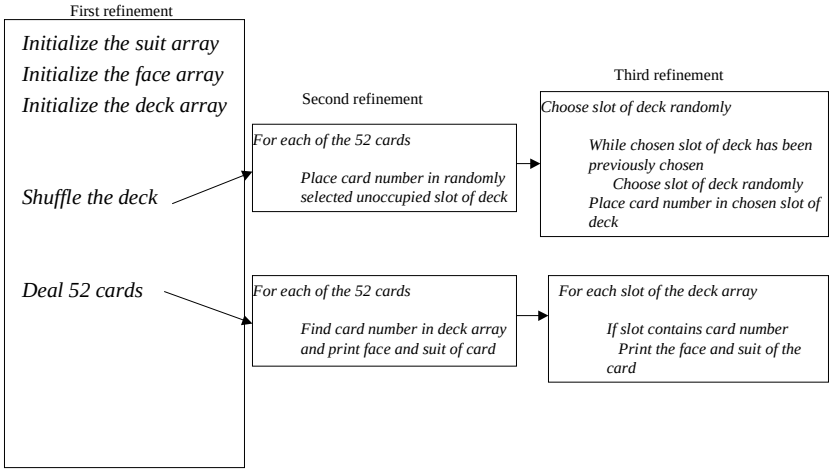
- Card shuffling program
  - Use array of pointers to strings
  - Use double scripted array (suit, face)



  - The numbers 1-52 go into the array - this is the order they are dealt

7.10 Case Study: A Card Shuffling and Dealing Simulation

- Pseudocode - Top level: *Shuffle and deal 52 cards*



```
1  /* Fig. 7.24: fig07_24.c
2     Card shuffling dealing program */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  void shuffle( int [][ 13 ] );
8  void deal( const int [][ 13 ], const char *[], const char *[] );
9
10 int main()
11 {
12    const char *suit[ 4 ] =
13       { "Hearts", "Diamonds", "Clubs", "Spades" };
14    const char *face[ 13 ] =
15       { "Ace", "Deuce", "Three", "Four",
16         "Five", "Six", "Seven", "Eight",
17         "Nine", "Ten", "Jack", "Queen", "King" };
18    int deck[ 4 ][ 13 ] = { 0 };
19
20    srand( time( 0 ) );
21
22    shuffle( deck );
23    deal( deck, face, suit );
24
25    return 0;
26 }
27
28 void shuffle( int wDeck[][ 13 ] )
29 {
30    int row, column, card;
31
32    for ( card = 1; card <= 52; card++ ) {
```

Outline

1. Initialize `suit` and `face` arrays

1.1 Initialize deck array

2. Call function `shuffle`

2.1 Call function `deal`

3. Define functions

```
33       do {
34          row = rand() % 4;
35          column = rand() % 13;
36       } while( wDeck[ row ][ column ] != 0 );
37
38       wDeck[ row ][ column ] = card;
39    }
40 }
41
42 void deal( const int wDeck[][ 13 ], const char *wFace[],
43            const char *wSuit[] )
44 {
45    int card, row, column;
46
47    for ( card = 1; card <= 52; card++ )
48
49       for ( row = 0; row <= 3; row++ )
50
51          for ( column = 0; column <= 12; column++ )
52
53             if ( wDeck[ row ][ column ] == card )
54                printf( "%5s of %-8s%c",
55                wFace[ column ], wSuit[ row ],
56                card % 2 == 0 ? '\n' : '\t' );
57 }
```

The numbers 1-52 are randomly placed into the **deck** array.

Searches **deck** for the **card** number, then prints the **face** and **suit**.

Outline

3. Define functions

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

Outline

Program Output

7.11 Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

7.11 Pointers to Functions (II)

- Example: bubblesort
  - Function **`bubble`** takes a function pointer
    - **`bubble`** calls this helper function
    - this determines ascending or descending sorting
  - The argument in **`bubblesort`** for the function pointer:

    ```
    bool ( *compare )( int, int )
    ```

    tells **`bubblesort`** to expect a pointer to a function that takes two **`int`**s and returns a **`bool`**.
  - If the parentheses were left out:

    ```
    bool *compare( int, int )
    ```

    - Declares a function that receives two integers and returns a pointer to a **`bool`**

```
1  /* Fig. 7.26: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5  void bubble( int [], const int, int (*)( int, int ) );
6  int ascending( int, int );
7  int descending( int, int );
8
9  int main()
10 {
11
12    int order,
13        counter,
14        a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    printf( "Enter 1 to sort in ascending order,\n"
17            "Enter 2 to sort in descending order: " );
18    scanf( "%d",  &order );
19    printf( "\nData items in original order\n" );
20
21    for ( counter = 0; counter < SIZE; counter++ )
22       printf( "%5d", a[ counter ] );
23
24    if ( order == 1 ) {
25       bubble( a, SIZE, ascending );
26       printf( "\nData items in ascending order\n" );
27    }
28    else {
29       bubble( a, SIZE, descending );
30       printf( "\nData items in descending order\n" );
31    }
32
```

Notice the function pointer parameter.

Outline

1. Initialize array.

2. Prompt for ascending or descending sorting.

2.1 Put appropriate function pointer into bubblesort.

2.2 Call bubble.

3. Print results.

```
33    for ( counter = 0; counter < SIZE; counter++ )
34       printf( "%5d", a[ counter ] );
35
36    printf( "\n" );
37
38    return 0;
39 }
40
41 void bubble( int work[], const int size,
42              int (*compare)( int, int ) )
43 {
44    int pass, count;
45
46    void swap( int *, int * );
47
48    for ( pass = 1; pass < size; pass++ )
49
50       for ( count = 0; count < size - 1; count++ )
51
52          if ( (*compare)( work[ count ], work[ count + 1 ] ) )
53             swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58    int temp;
59
60    temp = *element1Ptr;
61    *element1Ptr = *element2Ptr;
62    *element2Ptr = temp;
63 }
64
```

Outline

3.1 Define functions.

`ascending` and **`descending`** return **`true`** or **`false`**. **`bubble`** calls **`swap`** if the function call returns **`true`**.

Notice how function pointers are called using the dereferencing operator. The **`*`** is not required, but emphasizes that **`compare`** is a function pointer and not a function.

```
65 int ascending( int a, int b )
66 {
67    return b < a;   /* swap if b is less than a */
68 }
69
70 int descending( int a, int b )
71 {
72    return b > a;   /* swap if b is greater than a */
73 }
```

Outline

3.1 Define functions.

Program Output

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```


Chapter 8
Characters and Strings

Outline

8.1 Introduction

- Introduce some standard library functions
  - Easy string and character processing
  - Programs can process characters, strings, lines of text, and blocks of memory
- These techniques used to make
  - Word processors
  - Page layout software
  - Typesetting programs

8.2 Fundamentals of Strings and Characters

- Characters
  - Building blocks of programs
    - Every program is a sequence of meaningfully grouped characters
  - Character constant - an **`int`** value represented as a character in single quotes
    - **`'z'`** represents the integer value of **`z`**

8.2 Fundamentals of Strings and Characters (II)

- Strings
  - Series of characters treated as a single unit
    - Can include letters, digits, and certain special characters (`*`, `/`, `$`)
  - String literal (string constant) - written in double quotes
    - `"Hello"`
  - Strings are arrays of characters
    - String a pointer to first character
    - Value of string is the address of first character

8.2 Fundamentals of Strings and Characters (III)

- String declarations
  - Declare as a character array or a variable of type `char *`

```
char color[] = "blue";
char *colorPtr = "blue";
```

  - Remember that strings represented as character arrays end with `'\0'`
    - `color` has 5 elements

8.2 Fundamentals of Strings and Characters (IV)

- Inputting strings
  - Use **`scanf`**

    `scanf("%s", word);`
    - Copies input into **`word[]`**, which does not need **`&`** (because a string is a pointer)
  - Remember to leave space for **`'\0'`**

8.3 Character Handling Library

- Character Handling Library
  - Includes functions to perform useful tests and manipulations of character data
  - Each function receives a character (an **`int`**) or **`EOF`** as an argument

8.3 Character Handling Library (II)

- In **<ctype.h>**

Prototype	Description
`int isdigit( int c )`	Returns **true** if **c** is a digit and **false** otherwise.
`int isalpha( int c )`	Returns **true** if **c** is a letter and **false** otherwise.
`int isalnum( int c )`	Returns **true** if **c** is a digit or a letter and **false** otherwise.
`int isxdigit( int c )`	Returns **true** if **c** is a hexadecimal digit character and **false** otherwise.
`int islower( int c )`	Returns **true** if **c** is a lowercase letter and **false** otherwise.
`int isupper( int c )`	Returns **true** if **c** is an uppercase letter; **false** otherwise.
`int tolower( int c )`	If **c** is an uppercase letter, **tolower** returns **c** as a lowercase letter. Otherwise, **tolower** returns the argument unchanged.
`int toupper( int c )`	If **c** is a lowercase letter, **toupper** returns **c** as an uppercase letter. Otherwise, **toupper** returns the argument unchanged.
`int isspace( int c )`	Returns **true** if **c** is a white-space character—newline (**'\n'**), space (**' '**), form feed (**'\f'**), carriage return (**'\r'**), horizontal tab (**'\t'**), or vertical tab (**'\v'**)—and **false** otherwise
`int iscntrl( int c )`	Returns **true** if **c** is a control character and **false** otherwise.
`int ispunct( int c )`	Returns **true** if **c** is a printing character other than a space, a digit, or a letter and **false** otherwise.
`int isprint( int c )`	Returns **true** value if **c** is a printing character including space (**' '**) and **false** otherwise.
`int isgraph( int c )`	Returns **true** if **c** is a printing character other than space (**' '**) and **false** otherwise.

```
1  /* Fig. 8.2: fig08_02.c
2     Using functions isdigit, isalpha, isalnum, and isxdigit */
3  #include <stdio.h>
4  #include <ctype.h>
5
6  int main()
7  {
8     printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
9         isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
10        isdigit( '#' ) ? "# is a " :
11        "# is not a ", "digit" );
12    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
13        "According to isalpha:",
14        isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
15        isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
16        isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
17        isalpha( '4' ) ? "4 is a " :
18        "4 is not a ", "letter" );
19    printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
20        "According to isalnum:",
21        isalnum( 'A' ) ? "A is a " : "A is not a ",
22        "digit or a letter",
23        isalnum( '8' ) ? "8 is a " : "8 is not a ",
24        "digit or a letter",
25        isalnum( '#' ) ? "# is a " : "# is not a ",
26        "digit or a letter" );
27    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
28        "According to isxdigit:",
29        isxdigit( 'F' ) ? "F is a " : "F is not a ",
30        "hexadecimal digit",
31        isxdigit( 'J' ) ? "J is a " : "J is not a ",
32        "hexadecimal digit",
```

Outline

1. Load header
2. Perform tests
3. Print

```
33        isxdigit( '7' ) ? "7 is a " : "7 is not a ",
34        "hexadecimal digit",
35        isxdigit( '$' ) ? "$ is a " : "$ is not a ",
36        "hexadecimal digit",
37        isxdigit( 'f' ) ? "f is a " : "f is not a ",
38        "hexadecimal digit" );
39    return 0;
40 }
```

Outline

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Program Output

8.4 String Conversion Functions

- Conversion functions
  - In **<stdlib.h>** (general utilities library)
  - Convert strings of digits to integer and floating-

Prototype	Description
`double atof( const char *nPtr )`	Converts the string **nPtr** to **double**.
`int atoi( const char *nPtr )`	Converts the string **nPtr** to **int**.
`long atol( const char *nPtr )`	Converts the string **nPtr** to long **int**.
`double strtod( const char *nPtr, char **endPtr )`	Converts the string **nPtr** to **double.**
`long strtol( const char *nPtr, char **endPtr, int base )`	Converts the string **nPtr** to **long**.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converts the string **nPtr** to **unsigned long**.

```
1  /* Fig. 8.6: fig08_06.c
2     Using atof */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     double d;
9
10    d = atof( "99.0" );
11    printf( "%s%.3f\n%s%.3f\n",
12            "The string \"99.0\" converted to double is ", d,
13            "The converted value divided by 2 is ",
14            d / 2.0 );
15    return 0;
16 }
```

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

Outline

1. Initialize variable

2. Convert string
2.1 Assign to variable

3. Print

Program Output

8.5 Standard Input/Output Library Functions

- Functions in **<stdio.h>**
  - Used to manipulate character and string data

Function prototype	Function description
`int getchar( void );`	Inputs the next character from the standard input and returns it as an integer.
`char *gets( char *s );`	Inputs characters from the standard input into the array **s** until a newline or end-of-file character is encountered. A terminating null character is appended to the array.
`int putchar( int c );`	Prints the character stored in **c**.
`int puts( const char *s );`	Prints the string **s** followed by a newline character.
`int sprintf( char *s, const char *format, ... );`	Equivalent to **printf**, except the output is stored in the array **s** instead of printing it on the screen.
`int sscanf( char *s, const char *format, ... );`	Equivalent to **scanf**, except the input is read from the array **s** instead of reading it from the keyboard.

```
1  /* Fig. 8.13: fig08_13.c
2     Using gets and putchar */
3  #include <stdio.h>
4
5  int main()
6  {
7     char sentence[ 80 ];
8     void reverse( const char * const );
9
10    printf( "Enter a line of text:\n" );
11    gets( sentence );
12
13    printf( "\nThe line printed backwards is:\n" );
14    reverse( sentence );
15
16    return 0;
17 }
18
19 void reverse( const char * const sPtr )
20 {
21    if ( sPtr[ 0 ] == '\0' )
22       return;
23    else {
24       reverse( &sPtr[ 1 ] );
25       putchar( sPtr[ 0 ] );
26    }
27 }
```

reverse calls itself using substrings of the original string. When it reaches the **'\0'** character it prints using **putchar**

```
Enter a line of text:
Characters and Strings

The line printed backwards is:
sgnirtS dna sretcarahC
```

Outline

1. Initialize variables

2. Input

3. Print

3.1 Function definition (note on)

8.6 String Manipulation Functions of the String Handling Library

- Defined in **<string.h>**
- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Tokenize strings
  - Determine string length

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 )`	Copies string **s2** into array **s1**. The value of **s1** is returned.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copies at most **n** characters of string **s2** into array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, const char *s2 )`	Appends string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`char *strncat( char *s1, const char *s2, size_t n )`	Appends at most **n** characters of string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.

```
1  /* Fig. 8.19: fig08_19.c
2     Using strcat and strncat */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char s1[ 20 ] = "Happy ";
9     char s2[] = "New Year ";
10    char s3[ 40 ] = "";
11
12    printf( "s1 = %s\ns2 = %s\n", s1, s2 );
13    printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );
14    printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );
15    printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );
16    return 0;
17 }
```

```
s1 = Happy
s2 = New Year
strcat( s1, s2 ) = Happy New Year
strncat( s3, s1, 6 ) = Happy
strcat( s3, s1 ) = Happy Happy New Year
```


Outline

1. Initialize variables

2. Function calls

3. Print

Program Output

8.7 Comparison Functions of the String Handling Library

- Comparing strings
  - Computer compares numeric ASCII codes of characters in string
  - Appendix D has a list of character codes
- **`int strcmp( const char *s1, const char *s2 );`**
  - Compares string **`s1`** to **`s2`**
  - Returns a negative number (**`s1 < s2`**), zero (**`s1 == s2`**), or a positive number (**`s1 > s2`**)

8.8 Search Functions of the String Handling Library

Function prototype	Function description
`char *strchr( const char *s, int c );`	Locates the first occurrence of character **c** in string **s**. If **c** is found, a pointer to **c** in **s** is returned. Otherwise, a **NULL** pointer is returned.
`size_t strcspn( const char *s1, const char *s2 );`	Determines and returns the length of the initial segment of string **s1** consisting of characters not contained in string **s2**.
`size_t strspn( const char *s1, const char *s2 );`	Determines and returns the length of the initial segment of string **s1** consisting only of characters contained in string **s2**.
`char *strpbrk( const char *s1, const char *s2 );`	Locates the first occurrence in string **s1** of any character in string **s2**. If a character from string **s2** is found, a pointer to the character in string **s1** is returned. Otherwise, a **NULL** pointer is returned.
`char *strrchr( const char *s, int c );`	Locates the last occurrence of **c** in string **s**. If **c** is found, a pointer to **c** in string **s** is returned. Otherwise, a **NULL** pointer is returned.
`char *strstr( const char *s1, const char *s2 );`	Locates the first occurrence in string **s1** of string **s2**. If the string is found, a pointer to the string in **s1** is returned. Otherwise, a **NULL** pointer is returned.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—separated by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.

```
1  /* Fig. 8.27: fig08_27.c
2     Using strspn */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     const char *string1 = "The value is 3.14159";
9     const char *string2 = "aehi lsTuv";
10
11    printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
12            "string1 = ", string1, "string2 = ", string2,
13            "The length of the initial segment of string1",
14            "containing only characters from string2 = ",
15            strspn( string1, string2 ) );
16    return 0;
17 }
```

```
string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```


Outline

1. Initialize variables

2. Function calls

3. Print

Program Output

```
1  /* Fig. 8.29: fig08_29.c
2     Using strtok */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char string[] = "This is a sentence with 7 tokens";
9     char *tokenPtr;
10
11    printf( "%s\n%s\n\n%s\n",
12            "The string to be tokenized is:", string,
13            "The tokens are:" );
14
15    tokenPtr = strtok( string, " " );
16
17    while ( tokenPtr != NULL ) {
18       printf( "%s\n", tokenPtr );
19       tokenPtr = strtok( NULL, " " );
20    }
21
22    return 0;
23 }
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Outline

1. Initialize variables

2. Function calls

3. Print

Program Output

8.9 Memory Functions of the String-handling Library

- Memory Functions
  - Manipulate, compare, and search blocks of memory
  - Can manipulate any block of data
- Pointer parameters are **`void *`**
  - Any pointer can be assigned to **`void *`**, and vice versa
  - **`void *`** cannot be dereferenced
    - Each function receives a size argument specifying the number of bytes (characters) to process

8.9 Memory Functions of the String-handling Library (II)

"Object" refers to a block of data

Prototype	Description
`void *memcpy( void *s1, const void *s2, size_t n )`	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. A pointer to the resulting object is returned.
`void *memmove( void *s1, const void *s2, size_t n )`	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The copy is performed as if the characters are first copied from the object pointed to by **s2** into a temporary array, and then copied from the temporary array into the object pointed to by **s1**. A pointer to the resulting object is returned.
`int memcmp( const void *s1, const void *s2, size_t n )`	Compares the first **n** characters of the objects pointed to by **s1** and **s2**. The function returns **0**, less than **0**, or greater than **0** if **s1** is equal to, less than or greater than **s2**, respectively.
`void *memchr(const void *s, int c, size_t n )`	Locates the first occurrence of **c** (converted to **unsigned char**) in the first **n** characters of the object pointed to by **s**. If **c** is found, a pointer to **c** in the object is returned. Otherwise, **0** is returned.
`void *memset( void *s, int c, size_t n )`	Copies **c** (converted to **unsigned char**) into the first **n** characters of the object pointed to by **s**. A pointer to the result is returned.

```
1  /* Fig. 8.32: fig08_32.c
2     Using memmove */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char x[] = "Home Sweet Home";
9
10    printf( "%s%s\n",
11            "The string in array x before memmove is: ", x );
12    printf( "%s%s\n",
13            "The string in array x after memmove is:  ",
14            memmove( x, &x[ 5 ], 10 ) );
15
16    return 0;
17 }
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
```

Outline

1. Initialize variables

2. Function calls

3. Print

Program Output

8.10 Other Functions of the String Handling Library

- **`char *strerror(int errornum );`**
  - Creates a system-dependent error message based on **`errornum`**
  - Returns a pointer to the string

- **`size_t strlen(const char *s );`**
  - Returns the number of characters (before **`NULL`**) in string **`s`**

```
1  /* Fig. 8.37: fig08_37.c
2     Using strerror */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     printf( "%s\n", strerror( 2 ) );
9     return 0;
10 }
```

```
No such file or directory
```

Outline

1. Function call

2. Print

Program Output

Chapter 10 - Structures, Unions, Bit Fields, 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 *`** - **`face`** and **`suit`**

10.2 Structure Definitions (II)

- Struct information
  - A struct cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - Structure definition does not reserve space in memory
  - Creates a new data type that is 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 (III)

- 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;
```

  - Or:

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


10.4 Accessing Members of Structures

- Accessing structure members
  - Dot operator (`.`) - use with structure variable name

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

  - Arrow operator (`->`) - use with pointers to structure variables

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

`myCardPtr->suit` 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 Card *CardPtr;
```

  - Defines a new type name **`CardPtr`** as a synonym for type **`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

```
1  /* Fig. 10.3: fig10_03.c
2     The card shuffling and dealing program using structures */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  struct card {
8     const char *face;
9     const char *suit;
10 };
11
12 typedef struct card Card;
13
14 void fillDeck( Card * const, const char *[],
15                const char *[] );
16 void shuffle( Card * const );
17 void deal( const Card * const );
18
19 int main()
20 {
21    Card deck[ 52 ];
22    const char *face[] = { "Ace", "Deuce", "Three",
23                           "Four", "Five",
24                           "Six", "Seven", "Eight",
25                           "Nine", "Ten",
26                           "Jack", "Queen", "King"};
27    const char *suit[] = { "Hearts", "Diamonds",
28                           "Clubs", "Spades"};
29
30    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[]`


```
31
32    fillDeck( deck, face, suit );
33    shuffle( deck );
34    deal( deck );
35    return 0;
36 }
37
38 void fillDeck( Card * const wDeck, const char * wFace[],
39                const char * wSuit[] )
40 {
41    int i;
42
43    for ( i = 0; i <= 51; i++ ) {
44       wDeck[ i ].face = wFace[ i % 13 ];
45       wDeck[ i ].suit = wSuit[ i / 13 ];
46    }
47 }
48
49 void shuffle( Card * const wDeck )
50 {
51    int i, j;
52    Card temp;
53
54    for ( i = 0; i <= 51; i++ ) {
55       j = rand() % 52;
56       temp = wDeck[ i ];
57       wDeck[ i ] = wDeck[ j ];
58       wDeck[ j ] = temp;
59    }
60 }
```

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

2. Randomize `Deck`

2.1 `shuffle`

2.2 `deal`

3. Function definitions

```
61
62 void deal( const Card * const wDeck )
63 {
64    int i;
65
66    for ( i = 0; i <= 51; i++ )
67       printf( "%5s of %-8s%c", wDeck[ i ].face,
68               wDeck[ i ].suit,
69               ( i + 1 ) % 2 ? '\t' : '\n' );
70 }
```

Cycle through array and print out data.

Outline

definitions


```
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
```

Outline

Program Output

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 {
```


10.8 Unions (II)

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

```
1  /* Fig. 10.5: fig10_05.c
2     An example of a union */
3  #include <stdio.h>
4
5  union number {
6     int x;
7     double y;
8  };
9
10 int main()
11 {
12    union number value;
13
14    value.x = 100;
15    printf( "%s\n%s\n%s%d\n%s%f\n\n",
16            "Put a value in the integer member",
17            "and print both members.",
18            "int:   ", value.x,
19            "double:\n", value.y );
20
21    value.y = 100.0;
22    printf( "%s\n%s\n%s%d\n%s%f\n",
23            "Put a value in the floating member",
24            "and print both members.",
25            "int:   ", value.x,
26            "double:\n", value.y );
27    return 0;
28 }
```

Outline

1. Define union

1.1 Initialize variables

2. Set variables

3. Print


```
Put a value in the integer member
and print both members.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000000.00000

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

Outline

Program Output

10.11 Enumerations

- Enumeration
  - Set of integers represented by identifiers
  - Enumeration constants - like symbolic constants whose values automatically set
    - Values start at **0** and are incremented by **1**
    - Values can be set explicitly with **=**
    - Need unique constant names
  - Declare variables as usual
    - Enumeration variables can *only* assume their enumeration constant values (not the integer representations)

10.11 Enumerations (II)

- Example:

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

  - Starts at **1**, increments by **1**

```
1  /* Fig. 10.18: fig10_18.c
2     Using an enumeration type */
3  #include <stdio.h>
4
5  enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
6                JUL, AUG, SEP, OCT, NOV, DEC };
7
8  int main()
9  {
10    enum months month;
11    const char *monthName[] = { "", "January", "February",
12                                "March", "April", "May",
13                                "June", "July", "August",
14                                "September", "October",
15                                "November", "December" };
16
17    for ( month = JAN; month <= DEC; month++ )
18       printf( "%2d%11s\n", month, monthName[ month ] );
19
20    return 0;
21 }
```

Outline

1. Define enumeration

1.1 Initialize variable

2. Loop

2.1 Print

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**.

```
1  /* Fig. 10.9: fig10_09.c
2     Using the bitwise AND, bitwise inclusive OR, bitwise
3     exclusive OR and bitwise complement operators */
4  #include <stdio.h>
5
6  void displayBits( unsigned );
7
8  int main()
9  {
10    unsigned number1, number2, mask, setBits;
11
12    number1 = 65535;
13    mask = 1;
14    printf( "The result of combining the following\n" );
15    displayBits( number1 );
16    displayBits( mask );
17    printf( "using the bitwise AND operator & is\n" );
18    displayBits( number1 & mask );
19
20    number1 = 15;
21    setBits = 241;
22    printf( "\nThe result of combining the following\n" );
23    displayBits( number1 );
24    displayBits( setBits );
25    printf( "using the bitwise inclusive OR operator | is\n" );
26    displayBits( number1 | setBits );
27
28    number1 = 139;
29    number2 = 199;
30    printf( "\nThe result of combining the following\n" );
```

Outline

1. Function prototype

1.1 Initialize variables

2. Function calls

2.1 Print


```
31    displayBits( number1 );
32    displayBits( number2 );
33    printf( "using the bitwise exclusive OR operator ^ is\n" );
34    displayBits( number1 ^ number2 );
35
36    number1 = 21845;
37    printf( "\nThe one's complement of\n" );
38    displayBits( number1 );
39    printf( "is\n" );
40    displayBits( ~number1 );
41
42    return 0;
43 }
44
45 void displayBits( unsigned value )
46 {
47    unsigned c, displayMask = 1 << 31;
48
49    printf( "%7u = ", value );
50
51    for ( c = 1; c <= 32; c++ ) {
52       putchar( value & displayMask ? '1' : '0' );
53       value <<= 1;
54
55       if ( c % 8 == 0 )
56          putchar( ' ' );
57    }
58
59    putchar( '\n' );
60 }
```

Outline

2.1 Print

3. Function definition

MASK created with only one set bit, i.e.

(**10000000 00000000 00000000 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.

```
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
```

Outline

Program Output

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 (II)

- 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