مبانی برنامه نویسی
به زبان سی

۳۰ آذر، ۲ و ۴ دی ۱۳۹۹

جلسه بیست، بیست و یک و بیست و دو

ملکی مجد

مباحث این هفته:

- اشاره گر : عبارت ها و محاسبات
- ارتباط بین اشاره گر و آرایه
- آرایه ای از اشاره گر ها
- بررسی موردی (جابه جایی کارت ها)
- اشاره گر به تابع

- جلسه ۴ دی میان ترم برگزار خواهد شد.

7.8 Pointer Expressions and Pointer Arithmetic

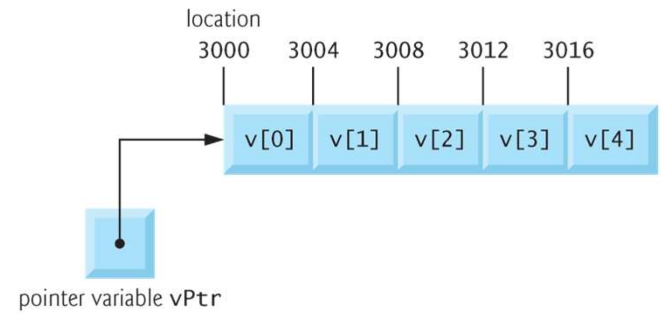
- Pointers are valid operands in
  - arithmetic expressions,
  - assignment expressions and
  - comparison expressions.

- not all the operators normally used in these expressions are valid in conjunction with pointer variables.
- A limited set of arithmetic operations may be performed on pointers.

A pointer may be incremented (++) or decremented (--),
an integer may be added to a pointer (+ or +=),
an integer may be subtracted from a pointer (- or -=) and
one pointer may be subtracted from another.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

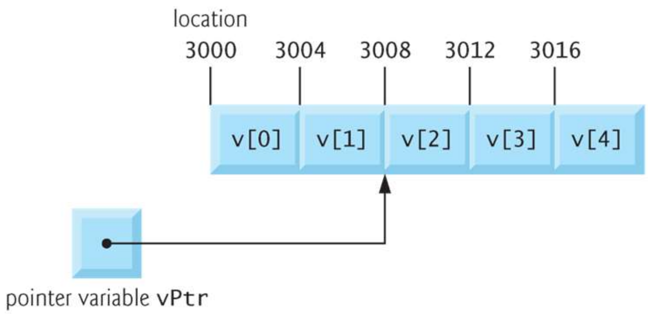
- Assume
  - array `int v[5]` has been defined and its first element is at location `3000` in memory.
  - pointer `vPtr` has been initialized to point to `v[0]`
    the value of `vPtr` is `3000`.
  - a machine with 4-byte integers.



- When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.
  - The number of bytes depends on the object's data type.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- In conventional arithmetic, `3000 + 2` yields the value `3002`.
  - This is normally not the case with pointer arithmetic.



- For example, the statement
  - `vPtr += 2;`

  would produce `3008 (3000 + 2 * 4)`
- In the array `v`, `vPtr` would now point to `v[2]`

Portability Tip 7.3

Most computers today have 2-byte or 4-byte integers. Some of the newer machines use 8-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

If an integer is stored in 2 bytes of memory,

then what is the result of the preceding calculation ?

```
vPtr = V = &V[0] = 3000
vPtr += 2
vPtr = ??
```

If an integer is stored in 2 bytes of memory,
then what is the result of the preceding calculation ?

```
vPtr = V = &V[0] = 3000
vPtr += 2
vPtr = ??
```

memory location 3004 (3000 + 2 * 2).

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type.
- When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- If `vPtr` had been incremented to `3016`, which points to `v[4]`, the statement
  - `vPtr -= 4;`

  would set vPtr back to 3000—the beginning of the array.

- If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used.
  - Either of the statements
    - ```
      ++vPtr;
      vPtr++;
      ```

    increments the pointer to point to the next location in the array.
  - Either of the statements
    - ```
      --vPtr;
      vPtr--;
      ```

    decrements the pointer to point to the previous element of the array.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Pointer variables may be subtracted from one another.
- For example, if `vPtr` contains the location `3000`, and `v2Ptr` contains the address `3008`, the statement
  - `x = v2Ptr - vPtr;`

  would assign to `x` the number of array elements from `vPtr` to `v2Ptr`, in this case `2` (not 8).

Pointer arithmetic is meaningless unless performed on an array.

We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- A pointer can be assigned to another pointer if both have the same type.

- The exception to this rule is the pointer to void (i.e., `void *`), which is a generic pointer that can represent any pointer type.
  - All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type.
  - In both cases, a cast operation is not required.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- A pointer to `void` cannot be dereferenced.

- Consider this: The compiler knows that a pointer to `int` refers to 4 bytes of memory on a machine with 4-byte integers, but a pointer to `void` simply contains a memory location for an unknown data type—the precise number of bytes to which the pointer refers is not known by the compiler.
- The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array.
- Pointer comparisons compare the addresses stored in the pointers.
- A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-numbered element of the array than the other pointer does.
- A common use of pointer comparison is determining whether a pointer is `NULL`.

7.9 Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An array name can be thought of as a constant pointer.

- Pointers can be used to do any operation involving array subscripting.
  - Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined.
  - we can set `bPtr` equal to the address of the first element in array `b` with the statement
    - `bPtr = b;`
  - This statement is equivalent to
    - `bPtr = &b[ 0 ];`
  - Array element `b[3]` can alternatively be referenced with the pointer expression
    - `*( bPtr + 3 )`
    - The `3` in the above expression is the offset to the pointer.

7.9 Relationship between Pointers and Arrays (Cont.)

- When the pointer points to the beginning of an array, the offset indicates which element of the array should be referenced, and the offset value is identical to the array subscript.
  - The preceding notation is referred to as pointer/offset notation.

7.9 Relationship between Pointers and Arrays (Cont.)

- The parentheses are necessary because the precedence of `*` is higher than the precedence of `+`.
  - Without the parentheses, the above expression would add `3` to the value of the expression `*bPtr` (i.e., `3` would be added to `b[0]`, assuming `bPtr` points to the beginning of the array).

7.9 Relationship between Pointers and Arrays (Cont.)

- Just as the array element can be referenced with a pointer expression, the address
  - `&b[ 3 ]`

  can be written with the pointer expression
  - `bPtr + 3`

- The array itself can be treated as a pointer and used in pointer arithmetic.
- For example, the expression
  - `*( b + 3 )`

  also refers to the array element b[3].

- Pointers can be subscripted exactly as arrays can.
- For example, if `bPtr` has the value `b`, the expression
  - `bPtr[ 1 ]`

  refers to the array element `b[1]`.
- This is referred to as pointer/subscript notation.

7.9 Relationship between Pointers and Arrays (Cont.)

- An array name is essentially a constant pointer; it always points to the beginning of the array.
- Thus, the expression
  - `b += 3`

  is **invalid** because it attempts to modify the value of the array name with pointer arithmetic.

```c
   Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main( void )
{
   int b[] = { 10, 20, 30, 40 }; /* initialize array b */
   int *bPtr = b; /* set bPtr to point to array b */
   int i; /* counter */
   int offset; /* counter */

   /* output array b using array subscript notation */
   printf( "Array b printed with:\nArray subscript notation\n" );

   /* loop through array b */
   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   } /* end for */

   /* output array b using array name and pointer/offset notation */
   printf( "\nPointer/offset notation where\n"
           "the pointer is the array name\n" );
```

```c
   /* loop through array b */
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
   } /* end for */

   /* output array b using bPtr and array subscript notation */
   printf( "\nPointer subscript notation\n" );

   /* loop through array b */
   for ( i = 0; i < 4; i++ ) {
      printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
   } /* end for */

   /* output array b using bPtr and pointer/offset notation */
   printf( "\nPointer/offset notation\n" );

   /* loop through array b */
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40
```

```
Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

7.9 Relationship between Pointers and Arrays (Cont.)

- To further illustrate the interchangeability of arrays and pointers, let's look at the two string-copying functions—`copy1` and `copy2`—in the program of Fig. 7.21.
- Both functions copy a string (possibly a character array) into a character array.
- After a comparison of the function prototypes for `copy1` and `copy2`, the functions appear identical.

`copy1` and `copy2` accomplish the same task; however, they’re implemented differently.

```c
/* Fig. 7.21: fig07_21.c
   Copying a string using array notation and pointer notation. */
#include <stdio.h>

void copy1( char * const s1, const char * const s2 ); /* prototype */
void copy2( char *s1, const char *s2 ); /* prototype */

int main( void )
{
   char string1[ 10 ]; /* create array string1 */
   char *string2 = "Hello"; /* create a pointer to a string */
   char string3[ 10 ]; /* create array string3 */
   char string4[] = "Good Bye"; /* create a pointer to a string */

   copy1( string1, string2 );
   printf( "string1 = %s\n", string1 );

   copy2( string3, string4 );
   printf( "string3 = %s\n", string3 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 1 of 2.)

```c

/* copy s2 to s1 using array notation */
void copy1( char * const s1, const char * const s2 )
{
   int i; /* counter */

   /* loop through strings */
   for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
      ; /* do nothing in body */
   } /* end for */
} /* end function copy1 */

/* copy s2 to s1 using pointer notation */
void copy2( char *s1, const char *s2 )
{
   /* loop through strings */
   for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
      ; /* do nothing in body */
   } /* end for */
} /* end function copy2 */
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 2 of 2.)

7.10 Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
- Each entry in the array is a string,
  - but in C a string is essentially a pointer to its first character.

- So each entry in an array of strings is actually a pointer to the first character of a string.
  - `const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };`

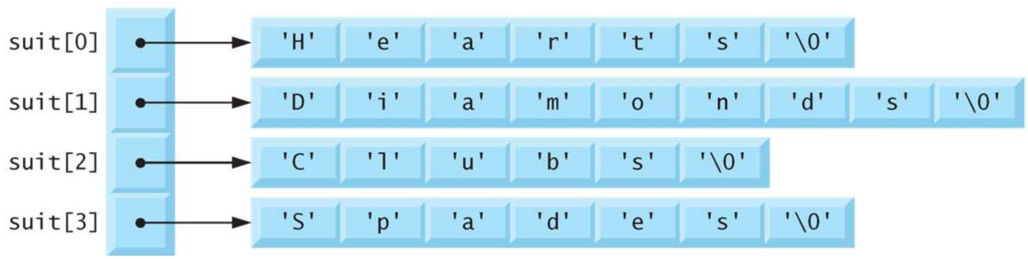
7.10 Arrays of Pointers (Cont.)

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

- The `suit[4]` portion of the definition indicates an array of 4 elements.
- The `char *` portion of the declaration indicates that each element of array `suit` is of type “pointer to `char`.”
- Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified.
- The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`.
- Each is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes.

7.10 Arrays of Pointers (Cont.)

- Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array (Fig. 7.22).



7.10 Arrays of Pointers (Cont.)

- The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name.
- Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string.
- Therefore, considerable memory could be wasted when a large number of strings were being stored with most strings shorter than the longest string.

7.11 Case Study: Card Shuffling and Dealing Simulation

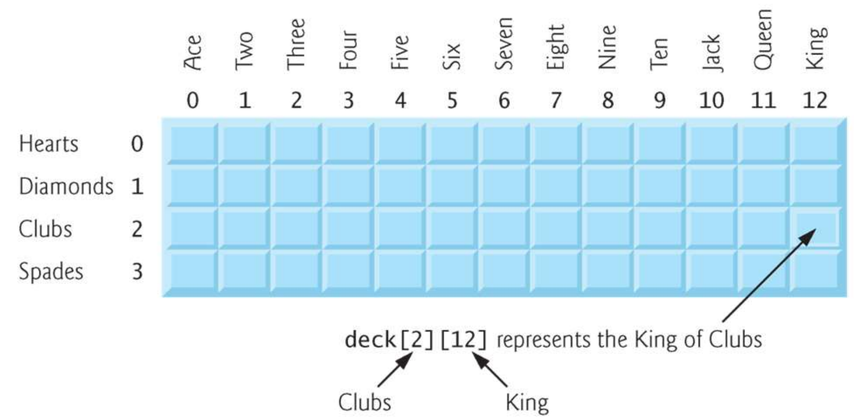
- The random number generation is used to develop a card shuffling and dealing simulation program.
  - This program can then be used to implement programs that play specific card games.

- Suboptimal shuffling and dealing algorithms are used intentionally.
  - You can develop more efficient algorithms as exercises in Chapter 10

- Using the top-down, stepwise refinement approach,

a program is developed to shuffle a deck of 52 playing cards and then deal each of the 52 cards.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)



- 4-by-13 double-subscripted array `deck`
  - to represent the deck of playing cards.
  - The rows correspond to the suits
    - row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades.
  - The columns correspond to the face values of the cards—
    - 0 through 9 correspond to ace through ten, and columns 10 through 12 correspond to jack, queen and king.
- We shall load string array `suit` with character strings representing the four suits, and string array `face` with character strings representing the thirteen face values.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- This simulated deck of cards may be shuffled as follows.

1. First the array `deck` is cleared to zeros.
2. Then, a `row` (0–3) and a `column` (0–12) are each chosen at random.
3. The number 1 is inserted in array element `deck[row][column]` to indicate that this card is going to be the first one dealt from the shuffled deck.

➢This process continues with the numbers 2, 3, …, 52 being randomly inserted in the `deck` array to indicate which cards are to be placed second, third, …, and fifty-second in the shuffled deck.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- As the `deck` array begins to fill with card numbers, it's possible that a card will be selected twice
    - i.e., `deck[row] [column]` will be nonzero when it's selected.
  - This selection is simply ignored and other `rows` and `columns` are repeatedly chosen at random until an unselected card is found.

- Eventually, the numbers 1 through 52 will occupy the 52 slots of the `deck` array.
  - At this point, the deck of cards is fully shuffled.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- This shuffling algorithm **could execute indefinitely** if cards that have already been shuffled are repeatedly selected at random.
- This phenomenon is known as indefinite postponement.

- In the book exercises, a better shuffling algorithm is discussed to eliminate the possibility of indefinite postponement.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- To deal the first card, we search the array for `deck[row][column]` equal to `1`.
  - This is accomplished with a nested `for` statement that varies `row` from 0 to 3 and `column` from 0 to 12.

- What card does that element of the array correspond to?
  - The `suit` array has been preloaded with the four suits, so to get the suit, we print the character string `suit[row]`.
  - Similarly, to get the face value of the card, we print the character string `face[column]`.
  - We also print the character string `" of "`.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Let's proceed with the top-down, stepwise refinement process.
- The top is simply
  - *Shuffle and deal 52 cards*
- Our first refinement yields:
  - Initialize the suit array
    Initialize the face array
    Initialize the deck array
    Shuffle the deck
    Deal 52 cards

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- “Shuffle the deck” may be expanded as follows:
  - For each of the 52 cards
    Place card number in randomly selected
    unoccupied slot of deck
- “Deal 52 cards” may be expanded as follows:
  - For each of the 52 cards
    Find card number in deck array and print face and
    suit of card
  - Printing this information in the proper order enables us to print each card in the form `"King of Clubs"`, `"Ace of Diamonds"` and so on.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Incorporating these expansions yields our complete second refinement:
  - Initialize the suit array
    Initialize the face array
    Initialize the deck array

    For each of the 52 cards
    Place card number in randomly selected
    unoccupied slot of deck

    For each of the 52 cards
    Find card number in deck array and print face and
    suit of card

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- "*Place card number in randomly selected unoccupied slot of deck*"

  may be expanded as:

  - Choose slot of deck randomly

    While chosen slot of deck has been previously chosen
    Choose slot of deck randomly

    Place card number in chosen slot of deck

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- "*Find card number in deck array and print face and suit of card*" may be expanded as:

  - For each slot of the deck array
    If slot contains card number
     Print the face and suit of the card

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Incorporating these expansions yields our third refinement:
  - Initialize the suit array
    Initialize the face array
    Initialize the deck array

    For each of the 52 cards
    Choose slot of deck randomly

    While slot of deck has been previously chosen
    Choose slot of deck randomly

    Place card number in chosen slot of deck

    For each of the 52 cards
    For each slot of deck array
     If slot contains desired card number
     Print the face and suit of the card

```c
   Card shuffling dealing program */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* prototypes */
void shuffle( int wDeck[][ 13 ] );
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] );

int main( void )
{
   /* initialize suit array */
   const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };

   /* initialize face array */
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };

   /* initialize deck array */
   int deck[ 4 ][ 13 ] = { 0 };

   srand( time( 0 ) ); /* seed random-number generator */

   shuffle( deck ); /* shuffle the deck */
   deal( deck, face, suit ); /* deal the deck */
   return 0; /* indicates successful termination */
} /* end main */

```

```c
/* shuffle cards in deck */
void shuffle( int wDeck[][ 13 ] )
{
   int row; /* row number */
   int column; /* column number */
   int card; /* counter */

   /* for each of the 52 cards, choose slot of deck randomly */
   for ( card = 1; card <= 52; card++ ) {

      /* choose new random location until unoccupied slot found */
      do {
         row = rand() % 4;
         column = rand() % 13;
      } while( wDeck[ row ][ column ] != 0 ); /* end do...while */

      /* place card number in chosen slot of deck */
      wDeck[ row ][ column ] = card;
   } /* end for */
} /* end function shuffle */

```

```c
/* deal cards in deck */
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
   int card; /* card counter */
   int row; /* row counter */
   int column; /* column counter */

   /* deal each of the 52 cards */
   for ( card = 1; card <= 52; card++ ) {
      /* loop through rows of wDeck */

      for ( row = 0; row <= 3; row++ ) {

         /* loop through columns of wDeck for current row */
         for ( column = 0; column <= 12; column++ ) {

            /* if slot contains current card, display card */
            if ( wDeck[ row ][ column ] == card ) {
               printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
                  card % 2 == 0 ? '\n' : '\t' );
            } /* end if */
         } /* end for */
      } /* end for */
   } /* end for */
} /* end function deal */
```

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

Fig. 7.25 | Sample run of card dealing program.

7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- There's a weakness in the dealing algorithm.
- Once a match is found, the two inner `for` statements continue searching the remaining elements of `deck` for a match.

```
/* deal each of the 52 cards */
for ( card = 1; card <= 52; card++ ) {
   /* loop through rows of wDeck */

   for ( row = 0; row <= 3; row++ ) {

      /* loop through columns of wDeck for current row */
      for ( column = 0; column <= 12; column++ ) {

         /* if slot contains current card, display card */
         if ( wDeck[ row ][ column ] == card ) {
            printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
               card % 2 == 0 ? '\n' : '\t' );
         } /* end if */
      } /* end for */
   } /* end for */
} /* end for */
```

7.12 Pointers to Functions

- A pointer to a function contains the address of the function in memory.
  - we saw that an array name is really the address in memory of the first element of the array.
  - Similarly, a function name is really the starting address in memory of the code that performs the function's task.

- Pointers to functions can **be passed to functions**, **returned from functions**, stored in arrays and **assigned to other function pointers**.

7.12 Pointers to Functions (Cont.)

- To illustrate the use of pointers to functions,

  consider a modified version of the bubble sort program
- The new version consists of `main` and functions `bubble`, `swap`, `ascending` and `descending`.

- Function `bubbleSort` receives a pointer to a function
  - either function `ascending` or
  - function `descending`
- as an argument, in addition to an integer array and the size of the array.

7.12 Pointers to Functions (Cont.)

- The program prompts the user to choose whether the array should be sorted in ascending or in descending order.
  - If the user enters `1`, a pointer to function `ascending` is passed to function `bubble`, causing the array to be sorted into increasing order.
  - If the user enters `2`, a pointer to function `descending` is passed to function `bubble`, causing the array to be sorted into decreasing order.

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order; /* 1 for ascending order or 2 for descending order */
   int counter; /* counter */

   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );

   printf( "\nData items in original order\n" );
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 1 of 4.)

```c

    /* output original array */
    for ( counter = 0; counter < SIZE; counter++ ) {
       printf( "%5d", a[ counter ] );
    } /* end for */

    /* sort array in ascending order; pass function ascending as an
      argument to specify ascending sorting order */
    if ( order == 1 ) {
       bubble( a, SIZE, ascending );
       printf( "\nData items in ascending order\n" );
    } /* end if */
    else { /* pass function descending */
       bubble( a, SIZE, descending );
       printf( "\nData items in descending order\n" );
    } /* end else */

    /* output sorted array */
    for ( counter = 0; counter < SIZE; counter++ ) {
       printf( "%5d", a[ counter ] );
    } /* end for */

    printf( "\n" );
    return 0; /* indicates successful termination */
 } /* end main */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 2 of 4.)

```c

/* multipurpose bubble sort; parameter compare is a pointer to
   the comparison function that determines sorting order */
void bubble( int work[], const int size, int (*compare)( int a, int b ) )
{
   int pass; /* pass counter */
   int count; /* comparison counter */

   void swap( int *element1Ptr, int *element2ptr ); /* prototype */

   /* loop to control passes */
   for ( pass = 1; pass < size; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( count = 0; count < size - 1; count++ ) {

         /* if adjacent elements are out of order, swap them */
         if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
            swap( &work[ count ], &work[ count + 1 ] );
         } /* end if */
      } /* end for */
   } /* end for */
} /* end function bubble */

```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 3 of 4.)

```c
/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold; /* temporary holding variable */

   hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */

/* determine whether elements are out of order for an ascending
   order sort */
int ascending( int a, int b )
{
   return b < a; /* swap if b is less than a */
} /* end function ascending */

/* determine whether elements are out of order for a descending
   order sort */
int descending( int a, int b )
{
   return b > a; /* swap if b is greater than a */
} /* end function descending */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 4 of 4.)

```
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
```

Fig. 7.27 | The outputs of the bubble sort program in Fig. 7.26.

7.12 Pointers to Functions (Cont.)

- The following parameter appears in the function header for `bubble` (line 52)
    - `int (*compare)( int a, int b )`
  - This tells `bubble` to expect a parameter (`compare`) that is a pointer to a function that receives two integer parameters and returns an integer result.

- Parentheses are needed around `*compare` to group `*` with `compare` to indicate that `compare` is a pointer.
- If we had not included the parentheses, the declaration would have been
    - `int *compare( int a, int b )`

  which declares a function that receives two integers as parameters and returns a pointer to an integer.

7.12 Pointers to Functions (Cont.)

- The prototype for `bubble` could have been written as
  - `int (*)( int, int );`

  without the function-pointer name and parameter names.

- The function passed to `bubble` is called in an `if` statement (line 66) as follows:
  - `if ((*compare)( work[count], work[count + 1]))`
- Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function.

7.12 Pointers to Functions (Cont.)

- The call to the function **could have been made without dereferencing** the pointer as in
  - `if (compare(work[count], work[count + 1]))`

  which uses the pointer directly as the function name.
- calling a function through a pointer

  explicitly illustrates that `compare` is a pointer to a function that is dereferenced to call the function.

  - The second method of calling a function through a pointer makes it appear as though `compare` is an actual function.
  - This may be confusing to a user of the program who would like to see the definition of function `compare` and finds that it's never defined in the file.

7.12 Pointers to Functions (Cont.)

- A common use of function pointers is in text-based menu-driven systems.
- A user is prompted to select an option from a menu (possibly from 1 to 5) by typing the menu item's number.
- Each option is serviced by a different function.
- Pointers to each function are stored in an array of pointers to functions.
- The user's choice is used as a subscript in the array, and the pointer in the array is used to call the function.

```c
/* Fig. 7.28: fig07_28.c
   Demonstrating an array of pointers to functions */
#include <stdio.h>

/* prototypes */
void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
   /* initialize array of 3 pointers to functions that each take an
      int argument and return void */
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variable to hold user's choice */

   printf( "Enter a number between 0 and 2, 3 to end: " );
   scanf( "%d", &choice );

```

The definition is read beginning in the leftmost set of parentheses, “`f` is an array of `3` pointers to functions that each take an `int` as an argument and return `void`.”

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 3.)

```
    /* process user's choice */
    while ( choice >= 0 && choice < 3 ) {

       /* invoke function at location choice in array f and pass
          choice as an argument */
       (*f[ choice ])( choice );

       printf( "Enter a number between 0 and 2, 3 to end: ");
       scanf( "%d", &choice );
    } /* end while */

    printf( "Program execution completed.\n" );
    return 0; /* indicates successful termination */
} /* end main */

void function1( int a )
{
    printf( "You entered %d so function1 was called\n\n", a );
} /* end function1 */

void function2( int b )
{
    printf( "You entered %d so function2 was called\n\n", b );
} /* end function2 */
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 3.)

```c

void function3( int c )
{
   printf( "You entered %d so function3 was called\n\n", c );
} /* end function3 */
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 3 of 3.)