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.

Since the array name (without a subscript) is a pointer to the first element of the array, 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 taking the address of the array’s first element as follows:
  \[ bPtr = &b[0]; \]

• 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.

• 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.
• 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).
• 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
  \[ \ast ( b + 3 ) \]
also refers to the array element \( b[3] \).

• In general, all subscripted array expressions can be written with a pointer and an offset.

• In this case, pointer/offset notation was used with the name of the array as a pointer.

• The preceding statement does not modify the array name in any way; \( b \) still points to the first element in the array.

• 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.
• Remember that 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.
Common Programming Error 7.10

*Attempting to modify an array name with pointer arithmetic is a syntax error.*
Figure 7.20 uses the four methods we have discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, pointer subscripting, and pointer/offset with a pointer—to print the four elements of the integer array $b$. 
/* Fig. 7.20: fig07_20.cpp
   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" );
}

Fig. 7.20  |  Using four methods of referencing array elements. (Part 1 of 3.)
/* 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 */

Fig. 7.20  |  Using four methods of referencing array elements. (Part 2 of 3.)
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

**Fig. 7.20** Using four methods of referencing array elements. (Part 3 of 3.)
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.

They accomplish the same task; however, they’re implemented differently.
/* 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.)
/* 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.)
• Function `copy1` uses array subscript notation to copy the string in `s2` to the character array `s1`.
• The function defines counter variable `i` as the array subscript.
• The `for` statement header (line 29) performs the entire copy operation—it’s body is the empty statement.
• The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop.
• The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`.
• When the null character is encountered in `s2`, it’s assigned to `s1`, and the value of the assignment becomes the value assigned to the left operand (`s1`).
Relationship between Pointers and Arrays

- The loop terminates because the integer value of the null character is zero (false).
- Function `copy2` uses pointers and pointer arithmetic to copy the string in `s2` to the character array `s1`.
- Again, the `for` statement header (line 38) performs the entire copy operation.
- The header does not include any variable initialization.
- As in function `copy1`, the expression `(*s1 = *s2)` performs the copy operation.
- Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`. 
Relationship between Pointers and Arrays

- After the assignment in the condition, the pointers are incremented to point to the next element of array \( s1 \) and the next character of string \( s2 \), respectively.
- When the null character is encountered in \( s2 \), it’s assigned to the dereferenced pointer \( s1 \) and the loop terminates.
- The first argument to both \texttt{copy1} and \texttt{copy2} must be an array large enough to hold the string in the second argument.
- Otherwise, an error may occur when an attempt is made to write into a memory location that is not part of the array.
- Also, the second parameter of each function is declared as \texttt{const char *} (a constant string).
• In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are never modified.

• Therefore, the second parameter is declared to point to a constant value so that the principle of least privilege is enforced—neither function requires the capability of modifying the second argument, so neither function is provided with that capability.
• 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.
• Consider the definition of string array suit, which might be useful in representing a deck of cards.

Arrays of Pointers

- 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.
• The four strings are 7, 9, 6 and 7 characters long, respectively.

• 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).

• Each pointer points to the first character of its corresponding string.

• Thus, even though the suit array is fixed in size, it provides access to character strings of any length.

• This flexibility is one example of C’s powerful data-structuring capabilities.
Fig. 7.22 | Graphical representation of the suit array.
• 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.
Case Study: Card Shuffling and Dealing Simulation

• In this section, we use random number generation to develop a card shuffling and dealing simulation program.
• This program can then be used to implement programs that play specific card games.
• To reveal some subtle performance problems, we have intentionally used suboptimal shuffling and dealing algorithms.
• In this chapter’s exercises and in Chapter 10, we develop more efficient algorithms.
• Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards and then deal each of the 52 cards.
The top-down approach is particularly useful in attacking larger, more complex problems than we have seen.

We use 4-by-13 double-subscripted array `deck` to represent the deck of playing cards (Fig. 7.23).

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.
**Fig. 7.23** | Double-subscripted array representation of a deck of cards.
• This simulated deck of cards may be shuffled as follows.
• First the array `deck` is cleared to zeros.
• Then, a `row` (0–3) and a `column` (0–12) are each chosen at random.
• 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.
Case Study: Card Shuffling and Dealing Simulation

• 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.
• 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 exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.
Performance Tip 7.3

Sometimes an algorithm that emerges in a “natural” way can contain subtle performance problems, such as indefinite postponement. Seek algorithms that avoid indefinite postponement.
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".
• 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.

• 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
• “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
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
Case Study: Card Shuffling and Dealing Simulation

- “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

- “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
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
Case Study: Card Shuffling and Dealing Simulation

- This completes the refinement process.
- This program is more efficient if the shuffle and deal portions of the algorithm are combined so that each card is dealt as it’s placed in the deck.
- We have chosen to program these operations separately because normally cards are dealt after they’re shuffled (not while they’re shuffled).
Case Study: Card Shuffling and Dealing Simulation

• The card shuffling and dealing program is shown in Fig. 7.24, and a sample execution is shown in Fig. 7.25.

• Conversion specifier %s is used to print strings of characters in the calls to printf.

• The corresponding argument in the printf call must be a pointer to char (or a char array).

• The format specification "%5s of %-8s" (line 73) prints a character string right justified in a field of five characters followed by " of " and a character string left justified in a field of eight characters.

• The minus sign in %-8s signifies left justification.
/* Fig. 7.24: fig07_24.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 */

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

Fig. 7.24 | Card dealing program. (Part 1 of 4.)
/* 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 */

**Fig. 7.24**  |  Card dealing program. (Part 2 of 4.)
/* 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 */

Fig. 7.24 | Card dealing program. (Part 3 of 4.)
```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 for */
        } /* end for */
    } /* end for */
} /* end function deal */
```
<table>
<thead>
<tr>
<th>Nine of Hearts</th>
<th>Five of Clubs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queen of Spades</td>
<td>Three of Spades</td>
</tr>
<tr>
<td>Queen of Hearts</td>
<td>Ace of Clubs</td>
</tr>
<tr>
<td>King of Hearts</td>
<td>Six of Spades</td>
</tr>
<tr>
<td>Jack of Diamonds</td>
<td>Five of Spades</td>
</tr>
<tr>
<td>Seven of Hearts</td>
<td>King of Clubs</td>
</tr>
<tr>
<td>Three of Clubs</td>
<td>Eight of Hearts</td>
</tr>
<tr>
<td>Three of Diamonds</td>
<td>Four of Diamonds</td>
</tr>
<tr>
<td>Queen of Diamonds</td>
<td>Five of Diamonds</td>
</tr>
<tr>
<td>Six of Diamonds</td>
<td>Five of Hearts</td>
</tr>
<tr>
<td>Ace of Spades</td>
<td>Six of Hearts</td>
</tr>
<tr>
<td>Nine of Diamonds</td>
<td>Queen of Clubs</td>
</tr>
<tr>
<td>Eight of Spades</td>
<td>Nine of Clubs</td>
</tr>
<tr>
<td>Deuce of Clubs</td>
<td>Six of Clubs</td>
</tr>
<tr>
<td>Deuce of Spades</td>
<td>Jack of Clubs</td>
</tr>
<tr>
<td>Four of Clubs</td>
<td>Eight of Clubs</td>
</tr>
<tr>
<td>Four of Spades</td>
<td>Seven of Spades</td>
</tr>
<tr>
<td>Seven of Diamonds</td>
<td>Seven of Clubs</td>
</tr>
<tr>
<td>King of Spades</td>
<td>Ten of Diamonds</td>
</tr>
<tr>
<td>Jack of Hearts</td>
<td>Ace of Hearts</td>
</tr>
<tr>
<td>Jack of Spades</td>
<td>Ten of Clubs</td>
</tr>
<tr>
<td>Eight of Diamonds</td>
<td>Deuce of Diamonds</td>
</tr>
<tr>
<td>Ace of Diamonds</td>
<td>Nine of Spades</td>
</tr>
<tr>
<td>Four of Hearts</td>
<td>Deuce of Hearts</td>
</tr>
<tr>
<td>King of Diamonds</td>
<td>Ten of Spades</td>
</tr>
<tr>
<td>Three of Hearts</td>
<td>Ten of Hearts</td>
</tr>
</tbody>
</table>

**Fig. 7.25** | Sample run of card dealing program.
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. We correct this deficiency in this chapter’s exercises and in a Chapter 10 case study.