7

C Pointers

7.7 `sizeof` Operator

- `sizeof`
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals `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. 7.17: fig07_17.c
2     Demonstrating the sizeof operator */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    float f;
12    double d;
13    long double ld;
14    int array[ 20 ];  /* create array of 20 int elements */
15    int *ptr = array; /* create pointer to array */
16
```


Outline

fig07_17.c

(1 of 2)


```
17    printf( "     sizeof c = %d\tsizeof(char)  = %d"
18            "\n     sizeof s = %d\tsizeof(short) = %d"
19            "\n     sizeof i = %d\tsizeof(int) = %d"
20            "\n     sizeof l = %d\tsizeof(long) = %d"
21            "\n     sizeof f = %d\tsizeof(float) = %d"
22            "\n     sizeof d = %d\tsizeof(double) = %d"
23            "\n    sizeof ld = %d\tsizeof(long double) = %d"
24            "\n sizeof array = %d"
25            "\n   sizeof ptr = %d\n",
26          sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
27          sizeof( int ), sizeof l, sizeof( long ), sizeof f,
28          sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
29          sizeof( long double ), sizeof array, sizeof ptr );
30
31    return 0; /* indicates successful termination */
32
33 } /* end main */
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
 sizeof array = 80
   sizeof ptr = 4
```


Outline

fig07_17.c

(2 of 2)

Portability Tip 7.2

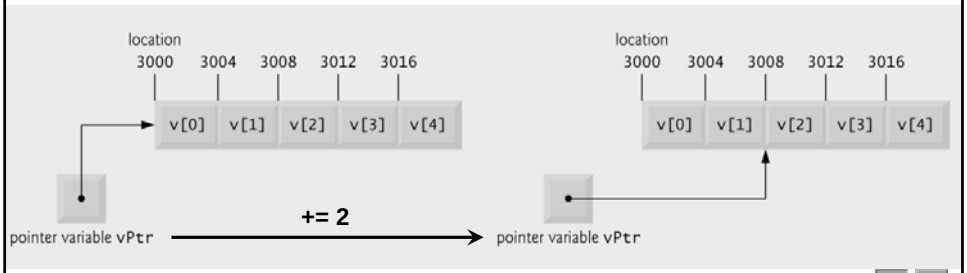
The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.

7.8 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**

7.8 Pointer Expressions and Pointer Arithmetic

- 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 the machine has 4 byte `int`s, so it points to address `3008`



7.8 Pointer Expressions and Pointer Arithmetic

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

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

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0`

Common Programming Errors

7.5: Using pointer arithmetic on a pointer that does not refer to an element in an array.

7.6: Subtracting or comparing two pointers that do not refer to elements in the same array.

7.7: Running off either end of an array when using pointer arithmetic.

7.8 Pointer Expressions and Pointer Arithmetic

- **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**

```
void *test;
char *cptr, c = 'a';
cptr = &c;
test = cptr;
printf("%c\n", *cptr);
printf("%c\n", *(char *) test);
```

Common Programming Errors

7.8: Assigning a pointer of one type to a pointer of another type if neither is of type void * is a syntax error.

7.9: Dereferencing a void * pointer is a syntax error.

7.9 Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

```
bPtr = b;
```

    - The array name (`b`) is actually the address of first element of the array `b[ 5 ]`

```
bPtr = &b[ 0 ]
```

    - Explicitly assigns `bPtr` to address of first element of `b`

7.9 Relationship Between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    Where 3 is called the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    Called pointer/subscript notation
    `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself
    `*( b + 3 )`

Outline

fig07_20.c

(1 of 3)

```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main( void )
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
```

Array subscript notation

Pointer/offset notation

Outline

fig07_20.c

(2 of 3)

```
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```

Pointer subscript notation

Pointer offset notation

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

(continued on next slide…)

Outline

fig07_20.c

(3 of 3)

(continued from previous slide…)

```
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
```

Outline

fig07_21.c

(1 of 2)

```
1  /* Fig. 7.21: fig07_21.c
2     Copying a string using array notation and pointer notation. */
3  #include <stdio.h>
4
5  void copy1( char * const s1, const char * const s2 ); /* prototype */
6  void copy2( char *s1, const char *s2 ); /* prototype */
7
8  int main( void )
9  {
10    char string1[ 10 ];          /* create array string1 */
11    char *string2 = "Hello";     /* create a pointer to a string */
12    char string3[ 10 ];          /* create array string3 */
13    char string4[] = "Good Bye"; /* create a pointer to a string */
14
15    copy1( string1, string2 );
16    printf( "string1 = %s\n", string1 );
17
18    copy2( string3, string4 );
19    printf( "string3 = %s\n", string3 );
20
21    return 0; /* indicates successful termination */
22
23 } /* end main */
24
```

Outline

fig07_21.c

(2 of 2)

```
25 /* copy s2 to s1 using array notation */
26 void copy1( char * const s1, const char * const s2 )
27 {
28    int i; /* counter */
29
30    /* loop through strings */
31    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
32       ; /* do nothing in body */
33    } /* end for */
34
35 } /* end function copy1 */
36
37 /* copy s2 to s1 using pointer notation */
38 void copy2( char *s1, const char *s2 )
39 {
40    /* loop through strings */
41    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42       ; /* do nothing in body */
43    } /* end for */
44
45 } /* end function copy2 */
```

Condition of **for** loop actually performs an action

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

7.10 Arrays of Pointers

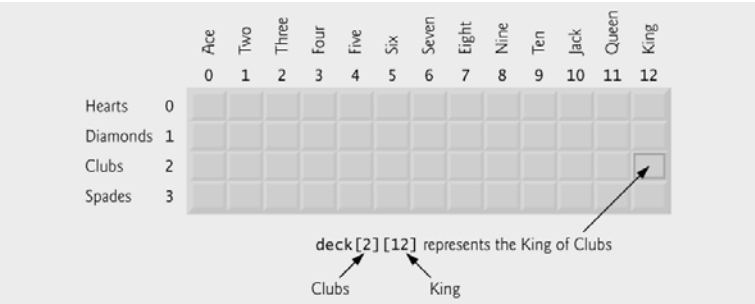
- Arrays can contain pointers
- For example: an array of strings

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

  - Strings are pointers to the first character
  - `char *` – each element of `suit` is a pointer to a `char`
  - The strings are not actually stored in the array `suit`, only pointers to the strings are stored

7.11 Case Study: Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use array of pointers to strings
  - Use double subscripted array (suit, face)
  - The numbers 1-52 go into the array
    - Representing the order in which the cards are dealt



7.11 Case Study: Card Shuffling and Dealing Simulation

- **Convert *shuffle the deck* to**
  - ***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***
- **Convert *deal 52 cards* to**
  - ***For each slot of the deck array***
    - ***If slot contains card number***
      - ***Print the face and suit of the card***

Outline

fig07_24.c

(1 of 4)

```
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  /* prototypes */
8  void shuffle( int wDeck[][ 13 ] );
9  void deal( const int wDeck[][ 13 ], const char *wFace[],
10            const char *wSuit[] );
11
12 int main( void )
13 {
14    /* initialize suit array */
15    const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
16
17    /* initialize face array */
18    const char *face[ 13 ] =
19       { "Ace", "Deuce", "Three", "Four",
20         "Five", "Six", "Seven", "Eight",
21         "Nine", "Ten", "Jack", "Queen", "King" };
22
```

suit and **face** arrays are arrays of pointers

```
23    /* initialize deck array */
24    int deck[ 4 ][ 13 ] = { 0 };
25
26    srand( time( 0 ) ); /* seed random-number generator */
27
28    shuffle( deck );
29    deal( deck, face, suit );
30
31    return 0; /* indicates successful termination */
32
33 } /* end main */
34
35 /* shuffle cards in deck */
36 void shuffle( int wDeck[][ 13 ] )
37 {
38    int row;    /* row number */
39    int column; /* column number */
40    int card;   /* counter */
41
42    /* for each of the 52 cards, choose slot of deck randomly */
43    for ( card = 1; card <= 52; card++ ) {
44
45       /* choose new random location until unoccupied slot found */
46       do {
47          row = rand() % 4;
48          column = rand() % 13;
49       } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
50
```

Outline

fig07_24.c

(2 of 4)

do…while loop selects a random spot for each card


```
51       /* place card number in chosen slot of deck */
52       wDeck[ row ][ column ] = card;
53    } /* end for */
54
55 } /* end function shuffle */
56
57 /* deal cards in deck */
58 void deal( const int wDeck[][ 13 ], const char *wFace[],
59            const char *wSuit[] )
60 {
61    int card;   /* card counter */
62    int row;    /* row counter */
63    int column; /* column counter */
64
65    /* deal each of the 52 cards */
66    for ( card = 1; card <= 52; card++ ) {
67       /* loop through rows of wDeck */
68
69       for ( row = 0; row <= 3; row++ ) {
70
71          /* loop through columns of wDeck for current row */
72          for ( column = 0; column <= 12; column++ ) {
```

Outline

fig07_24.c

(3 of 4)

```
73
74             /* if slot contains current card, display card */
75             if ( wDeck[ row ][ column ] == card ) {
76                printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
77                   card % 2 == 0 ? '\n' : '\t' );
78             } /* end if */
79
80          } /* end for */
81
82       } /* end for */
83
84    } /* end for */
85
86 } /* end function deal */
```


Outline

fig07_24.c

(4 of 4)


```
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
```


Outline

7.12 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**

7.12 Pointers to Functions

- **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:**

```
int ( *compare )( int a, int b )
```

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

```
int *compare( int a, int b )
```

    - **Defines a function that receives two integers and returns a pointer to a `int`**

Outline

fig07_26.c

(1 of 4)

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

bubble function takes a function pointer as an argument

Outline

fig07_26.c

(2 of 4)

```
30    /* sort array in ascending order; pass function ascending as an
31       argument to specify ascending sorting order */
32    if ( order == 1 ) {
33       bubble( a, SIZE, ascending );
34       printf( "\nData items in ascending order\n" );
35    } /* end if */
36    else { /* pass function descending */
37       bubble( a, SIZE, descending );
38       printf( "\nData items in descending order\n" );
39    } /* end else */
40
41    /* output sorted array */
42    for ( counter = 0; counter < SIZE; counter++ ) {
43       printf( "%5d", a[ counter ] );
44    } /* end for */
45
46    printf( "\n" );
47
48    return 0; /* indicates successful termination */
49
50 } /* end main */
51
```

depending on the user's choice, the **bubble** function uses either the **ascending** or **descending** function to sort the array

Outline

fig07_26.c

(3 of 4)

```
52 /* multipurpose bubble sort; parameter compare is a pointer to
53    the comparison function that determines sorting order */
54 void bubble( int work[], const int size, int (*compare)( int a, int b ) )
55 {
56    int pass;  /* pass counter */
57    int count; /* comparison counter */
58
59    void swap( int *element1Ptr, int *element2ptr ); /* prototype */
60
61    /* loop to control passes */
62    for ( pass = 1; pass < size; pass++ ) {
63
64       /* loop to control number of comparisons per pass */
65       for ( count = 0; count < size - 1; count++ ) {
66
67          /* if adjacent elements are out of order, swap them */
68          if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69             swap( &work[ count ], &work[ count + 1 ] );
70          } /* end if */
71
72       } /* end for */
73
74    } /* end for */
75
76 } /* end function bubble */
77
```

Note that what the program considers "out of order" is dependent on the function pointer that was passed to the **bubble** function

Outline

fig07_26.c

(4 of 4)

```
78 /* swap values at memory locations to which element1Ptr and
79    element2Ptr point */
80 void swap( int *element1Ptr, int *element2Ptr )
81 {
82    int hold; /* temporary holding variable */
83
84    hold = *element1Ptr;
85    *element1Ptr = *element2Ptr;
86    *element2Ptr = hold;
87 } /* end function swap */
88
89 /* determine whether elements are out of order for an ascending
90    order sort */
91 int ascending( int a, int b )
92 {
93    return b < a; /* swap if b is less than a */
94
95 } /* end function ascending */
96
97 /* determine whether elements are out of order for a descending
98    order sort */
99 int descending( int a, int b )
100 {
101    return b > a; /* swap if b is greater than a */
102
103 } /* end function descending */
```

Passing the **bubble** function **ascending** will point the program here

Passing the **bubble** function **descending** will point the program here

Outline

```
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
```

Outline

fig07_28.c

(1 of 3)

```
1  /* Fig. 7.28: fig07_28.c
2     Demonstrating an array of pointers to functions */
3  #include <stdio.h>
4
5  /* prototypes */
6  void function1( int a );
7  void function2( int b );
8  void function3( int c );
9
10 int main( void )
11 {
12    /* initialize array of 3 pointers to functions that each take an
13       int argument and return void */
14    void (*f[ 3 ])( int ) = { function1, function2, function3 };
15
16    int choice; /* variable to hold user's choice */
17
18    printf( "Enter a number between 0 and 2, 3 to end: " );
19    scanf( "%d", &choice );
20
```

Array of pointers to functions

Outline

```
21    /* process user's choice */
22    while ( choice >= 0 && choice < 3 ) {
23
24       /* invoke function at location choice in array f and pass
25          choice as an argument */
26       (*f[ choice ])( choice );
27
28       printf( "Enter a number between 0 and 2, 3 to end: ");
29       scanf( "%d", &choice );
30    } /* end while */
31
32    printf( "Program execution completed.\n" );
33
34    return 0; /* indicates successful termination */
35
36 } /* end main */
37
38 void function1( int a )
39 {
40    printf( "You entered %d so function1 was called\n\n", a );
41 } /* end function1 */
42
43 void function2( int b )
44 {
45    printf( "You entered %d so function2 was called\n\n", b );
46 } /* end function2 */
```

Function called is dependent on user's choice

fig07_28.c

(2 of 3)

Outline

```
47
48 void function3( int c )
49 {
50    printf( "You entered %d so function3 was called\n\n", c );
51 } /* 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.
```

fig07_28.c

(3 of 3)