C Programming Language

C Pointers

Introduction

- Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

Pointer Variable Definitions and Initialization

- Pointers are variables whose values are memory addresses.
- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an address of a variable that contains a specific value.
- Referencing a value through a pointer is called indirection.

Pointer Variable Definitions and Initialization (Cont.)

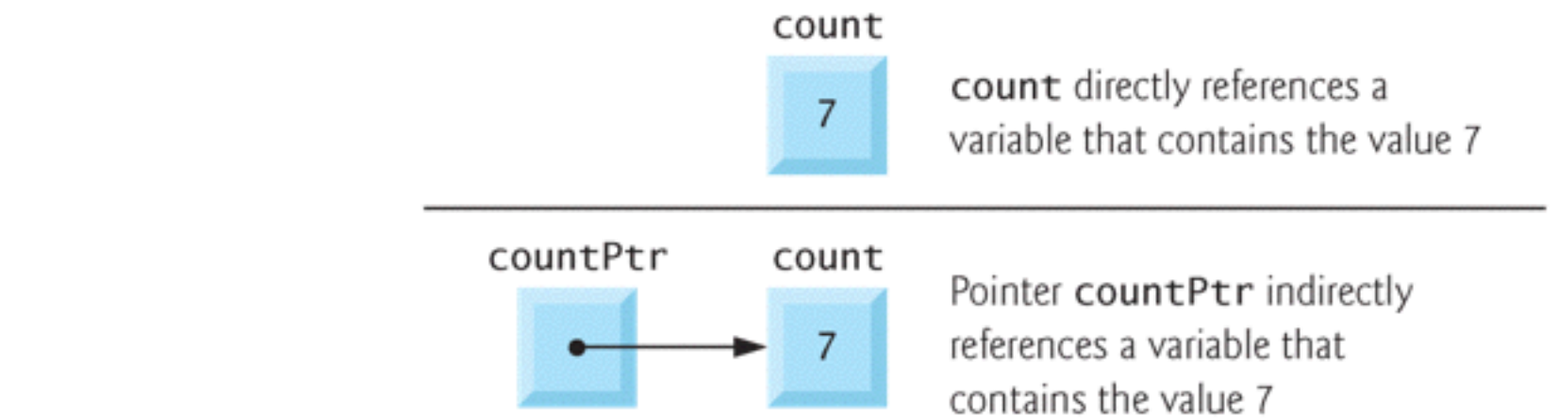


Fig. 7.1 | Directly and indirectly referencing a variable.

Pointer Variable Definitions and Initialization (Cont.)

- Pointer definition

```
int *countPtr;
```

  - **`countPtr`**: a pointer to **`int`**
  - `*`: indicates the variable being defined as a pointer
  - Pointers can be defined to point to objects of any type

Pointer Variable Definitions and Initialization (Cont.)

Common Programming Error 7.1

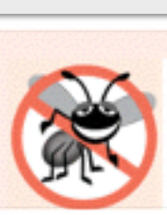
*The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare* `xPtr` *and* `yPtr` *as* `int` *pointers, use* `int *xPtr, *yPtr;`.

Common Programming Error 7.2

Include the letters `ptr` *in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.*

Pointer Variable Definitions and Initialization (Cont.)

- A pointer may be initialized to `NULL`, `0` or an address.
  - A pointer with the value `NULL` points to nothing.
  - Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.



Error-Prevention Tip 7.1

Initialize pointers to prevent unexpected results.

Pointer Operators

- The **&**, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions

```
int y = 5;
int *yPtr;

yPtr = &y;
```

  - Assigns the address of the variable y to pointer variable **`yPtr`**
  - Variable **`yPtr`** is then said to "point to" **`y`**.

Pointer Operators (Cont.)

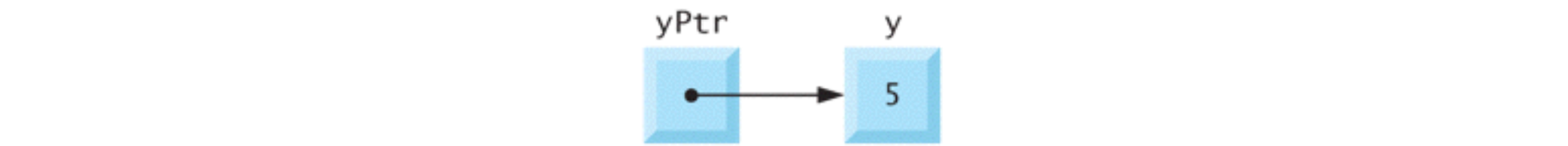


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Pointer Operators (Cont.)

- The operand of the address operator (**&**) must be a variable;
- The address operator (**&**) cannot be applied to constants, to expressions or to variables declared with the storage-class register.

Pointer Operators (Cont.)

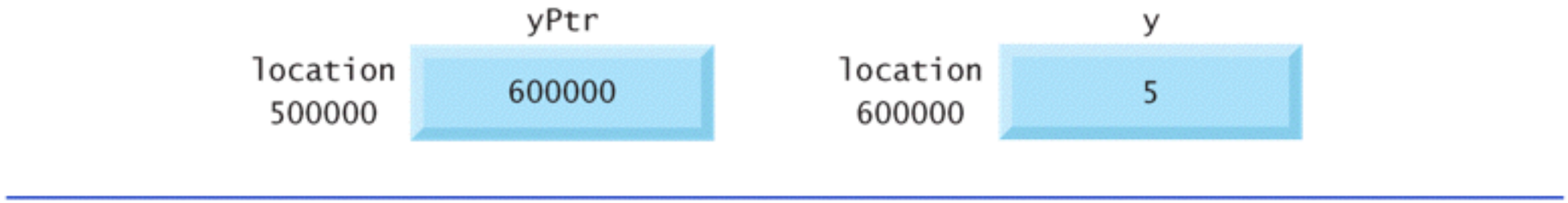


Fig. 7.3 | Representation of `y` and `yPtr` in memory.

Pointer Operators (Cont.)

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points.
- For example, the statement

```
printf( "%d", *yPtr );
```

  - prints the value of variable **y**, namely 5.
- Using * in this manner is called dereferencing a pointer.

Pointer Operators (Cont.)

- Example: fig07_04.c

```
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 complements of "
20            "each other\n&*aPtr = %p"...
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22    return 0; /* indicates successful termination */
```

declare a pointer to an integer

set the address of a to aPtr

use **&** to get address

use * to dereference a pointer

* and & are complements to each other

```
The address of a is 0x7fff5fbfe74c
The value of aPtr is 0x7fff5fbfe74c

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0x7fff5fbfe74c
*&aPtr = 0x7fff5fbfe74c
```

Pointer Operators (Cont.)

Operators	Associativity	Type
() []	left to right	highest
+ - ++ -- ! * & (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 7.5 | Operator precedence and associativity.

Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—call-by-value and call-by-reference.
- All arguments in C are passed by value.
- C provides the capabilities for simulating call-by-reference.
- In C, you use pointers and the indirection operator to simulate call-by-reference.

Passing Arguments to Functions by Reference (Cont.)

- This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

Passing Arguments to Functions by Reference (Cont.)

- Example: fig07_06.c

```
 5 int cubeByValue( int n ); /* prototype */
 6
 7 int main( void )
 8 {
 9    int number = 5; /* initialize number */
10
11    printf( "The original value of number is %d", number );
12
13    /* pass number by value to cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "\nThe new value of number is %d\n", number );
17    return 0; /* indicates successful termination */
18 } /* end main */
19
20 /* calculate and return cube of integer argument */
21 int cubeByValue( int n )
22 {
23    return n * n * n; /* cube local variable n and return result */
24 } /* end function cubeByValue */
```

call by value

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

Passing Arguments to Functions by Reference (Cont.)

- Example: fig07_07.c

```
 6 void cubeByReference( int *nPtr ); /* prototype */
 7
 8 int main( void )
 9 {
10    int number = 5; /* initialize number */
11
12    printf( "The original value of number is %d", number );
13
14    /* pass address of number to cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18    return 0; /* indicates successful termination */
19 } /* end main */
20
21 /* calculate cube of *nPtr; modifies variable number in main */
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; /* cube *nPtr */
25 } /* end function cubeByReference */
```

declare a function that can receive pointers as arguments

use & to pass the address of number

1. declare a pointer, `nPtr`, to receive the address

2. use * to dereference the pointer

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

Passing Arguments to Functions by Reference (Cont.)

Step 1: Before `main` calls `cubeByValue`:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

Passing Arguments to Functions by Reference (Cont.)

Step 3: After **cubeByValue** cubes parameter **n** and before **cubeByValue** returns to **main**:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

Step 4: After **cubeByValue** returns to **main** and before assigning the result to **number**:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Passing Arguments to Functions by Reference (Cont.)

Step 5: After **main** completes the assignment to **number**:

```
int main( void )                                    number
{                                                   125
   int number = 5;
      125                  125
   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
                                                    n
                                                    undefined
```

Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Passing Arguments to Functions by Reference (Cont.)

Step 1: Before `main` calls `cubeByReference`:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: undefined

Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: *call establishes this pointer*

Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Passing Arguments to Functions by Reference (Cont.)

Step 3: After *nPtr is cubed and before program control returns to main:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 125

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

called function modifies caller's variable

Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Using the `const` Qualifier with Pointers

- The `const` qualifier enables you to inform the compiler that the value of a particular variable should not be modified.

Using the `const` Qualifier with Pointers (Cont.)

Software Engineering Observation 7.1

The `const` *qualifier can be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software reduces debugging time and improper side effects, making a program easier to modify and maintain.*

Error-Prevention Tip 7.3

If a variable does not (or should not) change in the body of a function to which it's passed, the variable should be declared `const` *to ensure that it's not accidentally modified.*

Error-Prevention Tip 7.4

Before using a function, check its function prototype to determine if the function is able to modify the values passed to it.

Using the **`const`** Qualifier with Pointers (Cont.)

- There are 4 ways to pass a pointer to a function:
  - a non-constant pointer to non-constant data
  - a constant pointer to non-constant data
  - a non-constant pointer to constant data
  - a constant pointer to constant data
- Each of the four combinations provides different access privileges

Using the `const` Qualifier with Pointers (Cont.)

- The highest level of data access is granted by a non-constant pointer to non-constant data.
- In this case, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.
- A declaration for a non-constant pointer to non-constant data does not include **`const`**.
- Below we show a case of converting a string to uppercase by using a non-constant pointer to non-constant data

Using the **`const`** Qualifier with Pointers (Cont.)

- Example: fig07_10.c

```
 6 #include <ctype.h>
 7
 8 void convertToUppercase( char *sPtr ); /* prototype */
 9
10 int main( void )
11 {
12    char string[] = "characters and $32.98"; /* initialize char array */
13
14    printf( "The string before conversion is: %s", string );
15    convertToUppercase( string );
16    printf( "\nThe string after conversion is: %s\n", string );
17    return 0; /* indicates successful termination */
18 } /* end main */
```

because of **`islower()`** and

declare a string

pass a non-constant pointer to the function

```
21 void convertToUppercase( char *sPtr )
22 {
23    while ( *sPtr != '\0' ) { /* current character is not '\0' */
24
25       if ( islower( *sPtr ) ) { /* if character is lowercase, */
26          *sPtr = toupper( *sPtr ); /* convert to uppercase */
27       } /* end if */
28
29       ++sPtr; /* move sPtr to the next character */
30    } /* end while */
31 } /* end function convertToUppercase */
```

receive a pointer

if character is lowercase, convert to uppercase

move the pointer to the next character

Using the **`const`** Qualifier with Pointers (Cont.)

- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
- Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data.

Using the **`const`** Qualifier with Pointers (Cont.)

- Example: fig07_11.c

```
 7 void printCharacters( const char *sPtr );
 8
 9 int main( void )
10 {
11    /* initialize char array */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
15    printCharacters( string );
16    printf( "\n" );
17    return 0; /* indicates successful termination */
18 } /* end main */
19
20 /* sPtr cannot modify the character to which it points,
21    i.e., sPtr is a "read-only" pointer */
22 void printCharacters( const char *sPtr )
23 {
24    /* loop through entire string */
25    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
26       printf( "%c", *sPtr );
27    } /* end for */
28 } /* end function printCharacters */
```

`sPtr` is a pointer to a character constant

the character pointed by **`sPtr`** cannot be modified

Using the **`const`** Qualifier with Pointers (Cont.)

- Example: fig07_12.c

```
 6 void f( const int *xPtr ); /* prototype */
 7
 8 int main( void )
 9 {
10    int y; /* define y */
11
12    f( &y ); /* f attempts illegal modification */
13    return 0; /* indicates successful termination */
14 } /* end main */
15
16 /* xPtr cannot be used to modify the
17    value of the variable to which it points */
18 void f( const int *xPtr )
19 {
20    *xPtr = 100; /* error: cannot modify a const object */
21 } /* end function f */
```

`xPtr` is a pointer to an integer constant

invoke **`f( )`**

error: cannot modify a const object!!

```
fig07_12.c: In function ‘f’:
fig07_12.c:20: error: assignment of read-only location
```

Using the `const` Qualifier with Pointers (Cont.)

- Using pointers to constant data in this manner is an example of a time/space trade-off.
  - If memory is low and execution efficiency is a concern, use pointers.
  - If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege.
  - Remember that some systems do not enforce const well, so call-by-value is still the best way to prevent data from being modified.

Using the `const` Qualifier with Pointers (Cont.)

- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer.
- This is the default for an array name.
  - An array name is a constant pointer to the beginning of the array.
  - All data in the array can be accessed and changed by using the array name and array subscripting.

Using the **`const`** Qualifier with Pointers (Cont.)

- Example: fig07_13.c

```
 5 int main( void )
 6 {
 7    int x; /* define x */
 8    int y; /* define y */
 9
10    /* ptr is a constant pointer to an integer that can be modified
11       through ptr, but ptr always points to the same memory location */
12    int * const ptr = &x;
13
14    *ptr = 7; /* allowed: *ptr is not const */
15    ptr = &y; /* error: ptr is const; cannot assign new address */
16    return 0; /* indicates successful termination */
17 } /* end main */
```

`ptr` is a constant pointer to an integer; **`ptr`** is initialized with the address of **`x`**

error: **`ptr`** is const; cannot assign new address!!

```
fig07_13.c: In function ‘main’:
fig07_13.c:15: error: assignment of read-only variable ‘ptr’
```

Using the `const` Qualifier with Pointers (Cont.)

- The least access privilege is granted by a constant pointer to constant data.
- Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.
- This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array.

Using the **`const`** Qualifier with Pointers (Cont.)

- Example: fig07_14.c

```
 5 int main( void )
 6 {
 7    int x = 5; /* initialize x */
 8    int y; /* define y */
 9
10    /* ptr is a constant pointer to a constant integer. ptr always
11       points to the same location; the integer at that location
12       cannot be modified */
13    const int *const ptr = &x;
14
15    printf( "%d\n", *ptr );
16
17    *ptr = 7; /* error: *ptr is const; cannot assign new value */
18    ptr = &y; /* error: ptr is const; cannot assign new address */
19    return 0; /* indicates successful termination */
20 } /* end main */
```

`ptr` is a constant pointer to an integer constant

```
fig07_14.c: In function ‘main’:
fig07_14.c:17: error: assignment of read-only location
fig07_14.c:18: error: assignment of read-only variable ‘ptr’
```

error: **`*ptr`** is const; cannot assign new value
error: **`ptr`** is const; cannot assign new address

Bubble Sort Using Call-by-Reference

- Let's implement the bubble sort program by two functions—**`bubbleSort`** and **`swap`**.
  - Function `bubbleSort` sorts the array.
  - Function `swap` exchanges the array elements `array[j]` and `array[j + 1]`

Bubble Sort Using Call-by-Reference (Cont.)

- Example: fig07_15.c

```
7  void bubbleSort( int * const array, const int size ); /* prototype */
8
9  int main( void )
10 {
11    /* initialize array a */
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    int i; /* counter */
15
16    printf( "Data items in original order\n" );
17
18    /* loop through array a */
19    for ( i = 0; i < SIZE; i++ ) {
20       printf( "%4d", a[ i ] );
21    } /* end for */
22
23    bubbleSort( a, SIZE ); /* sort the array */
24
25    printf( "\nData items in ascending order\n" );
26
27    /* loop through array a */
28    for ( i = 0; i < SIZE; i++ ) {
29       printf( "%4d", a[ i ] );
30    } /* end for */
31
32    printf( "\n" );
33    return 0; /* indicates successful termination */
34 } /* end main */
```

`array` is a constant pointer to an integer

invoke **`bubbleSort( )`**

Bubble Sort Using Call-by-Reference (Cont.)

- Example: fig07_15.c

```
37 void bubbleSort( int * const array, const int size )
38 {
39    void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
40    int pass; /* pass counter */
41    int j; /* comparison counter */
42
43    /* loop to control passes */
44    for ( pass = 0; pass < size - 1; pass++ ) {
45
46       /* loop to control comparisons during each pass */
47       for ( j = 0; j < size - 1; j++ ) {
48
49          /* swap adjacent elements if they are out of order */
50          if ( array[ j ] > array[ j + 1 ] ) {
51             swap( &array[ j ], &array[ j + 1 ] );
52          } /* end if */
53       } /* end inner for */
54    } /* end outer for */
55 } /* end function bubbleSort */
```

use call-by-reference to swap these two elements

pass the addresses of two elements to swap

Bubble Sort Using Call-by-Reference (Cont.)

- Example: fig07_15.c

```
59 void swap( int *element1Ptr, int *element2Ptr )
60 {
61    int hold = *element1Ptr;
62    *element1Ptr = *element2Ptr;
63    *element2Ptr = hold;
64 } /* end function swap */
```

use a hold to put the temporary data

```
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
```

`sizeof` Operator

- C provides the special unary operator **`sizeof`** to determine the size in bytes of an array (or any other data type) during program compilation.

`sizeof` Operator (Cont.)

- Example: fig07_16.c

```
 6 size_t getSize( float *ptr ); /* prototype */
 7
 8 int main( void )
 9 {
10    float array[ 20 ]; /* create array */
11
12    printf( "The number of bytes in the array is %d"
13            "\nThe number of bytes returned by getSize is %d\n",
14            sizeof( array ), getSize( array ) );
15    return 0; /* indicates successful termination */
16 } /* end main */
17
18 /* return size of ptr */
19 size_t getSize( float *ptr )
20 {
21    return sizeof( ptr );
22 } /* end function getSize */
```

`size_t` is a type defined by the C standard as the integral type of the value returned by operator sizeof

return the size of the array pointed by **`ptr`**

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 8
```

`sizeof` Operator (Cont.)

- The number of elements in an array also can be determined with **`sizeof`**.
- For example, consider the following array definition:

```
double real[ 22 ];
```

- To determine the number of elements in the array, the following expression can be used:

```
sizeof( real ) / sizeof( real[ 0 ] )
```

`sizeof` Operator (Cont.)

- Type **`size_t`** is a type defined by the C standard as the integral type (unsigned or unsigned long) of the value returned by operator sizeof.
- Type **`size_t`** is defined in header **`<stddef.h>`** (which is included by several headers, such as **`<stdio.h>`**).
- [Note: If you attempt to compile Fig. 7.16 and receive errors, simply include **`<stddef.h>`** in your program.]

`sizeof` Operator (Cont.)

- Example: fig07_17.c

```c
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
    int array[ 20 ]; /* create array of 20 int elements */
    int *ptr = array; /* create pointer to array */

    printf( "    sizeof c = %d\tsizeof(char)  = %d"
            "\n    sizeof s = %d\tsizeof(short) = %d"
            "\n    sizeof i = %d\tsizeof(int) = %d"
            "\n    sizeof l = %d\tsizeof(long) = %d"
            "\n    sizeof f = %d\tsizeof(float) = %d"
            "\n    sizeof d = %d\tsizeof(double) = %d"
            "\n   sizeof ld = %d\tsizeof(long double) = %d"
            "\nsizeof array = %d"
            "\n  sizeof ptr = %d\n",
            sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
            sizeof( int ), sizeof l, sizeof( long ), sizeof f,
            sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
            sizeof( long double ), sizeof array, sizeof ptr );
```

```
    sizeof c = 1        sizeof(char)  = 1
    sizeof s = 2        sizeof(short) = 2
    sizeof i = 4        sizeof(int) = 4
    sizeof l = 8        sizeof(long) = 8
    sizeof f = 4        sizeof(float) = 4
    sizeof d = 8        sizeof(double) = 8
   sizeof ld = 16       sizeof(long double) = 16
sizeof array = 80
  sizeof ptr = 8
```

`sizeof` Operator (Cont.)

- Operator **`sizeof`** can be applied to any variable name, type or value (including the value of an expression).
- When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned.
- The parentheses used with sizeof are required if a type name with two words is supplied as its operand (such as long double or unsigned short).

Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- This section describes the operators that can have pointers as operands, and how these operators are used.

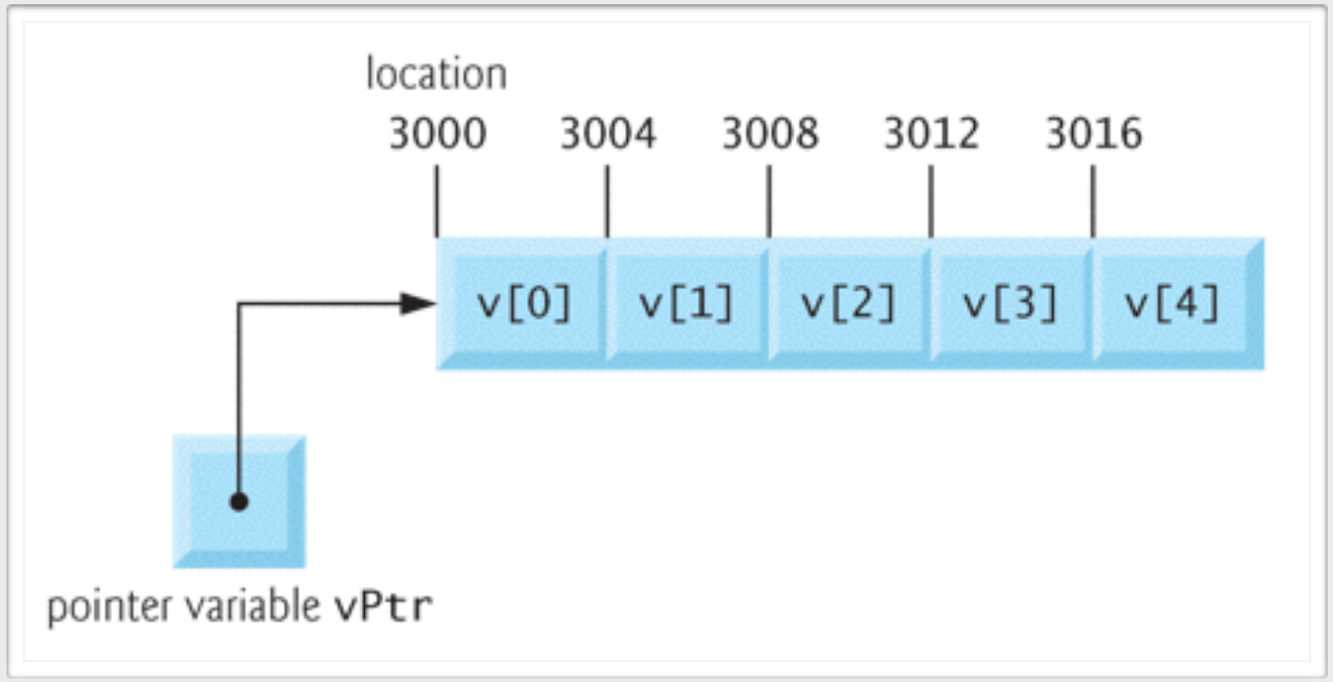
Pointer Expressions and Pointer Arithmetic (Cont.)

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.

Pointer Expressions and Pointer Arithmetic (Cont.)

- Assume that array **`int v[5]`** has been defined and its first element is at location 3000 in memory.
- Assume pointer **`vPtr`** has been initialized to point to **`v[0]`**—i.e., the value of **`vPtr`** is 3000.

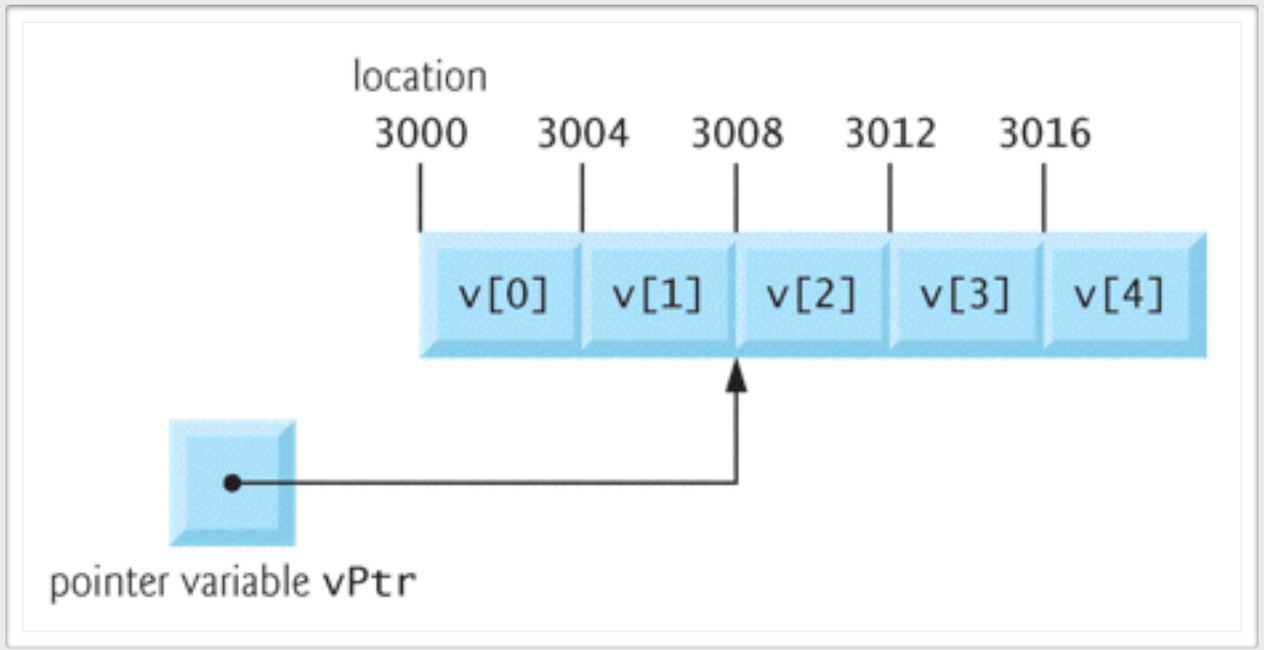


Pointer Expressions and Pointer Arithmetic (Cont.)

- The statement

```
vPtr += 2;
```

would produce 3008 (3000 + 2 * 4), assuming an integer is stored in 4 bytes of memory.

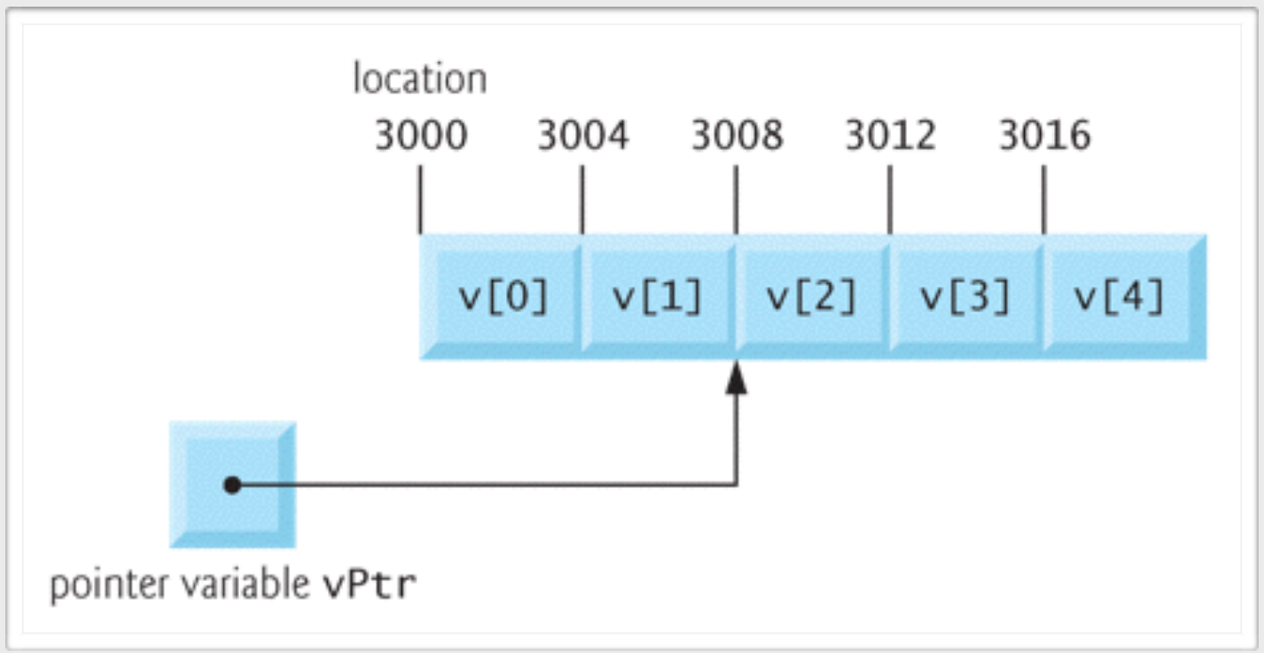


Pointer Expressions and Pointer Arithmetic (Cont.)

- 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.



Pointer Expressions and Pointer Arithmetic (Cont.)

- 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.

Pointer Expressions and Pointer Arithmetic (Cont.)

- 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 Expressions and Pointer Arithmetic (Cont.)

Common Programming Error 7.5

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

Common Programming Error 7.6

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

Common Programming Error 7.7

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

Pointer Expressions and Pointer Arithmetic (Cont.)

- Pointer to `void` (i.e., **`void *`**), which is a generic pointer that can represent any pointer type.
  - A pointer to void can be assigned a pointer of any type.
  - A pointer to void cannot be dereferenced.
- The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.

Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- 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;
```

Relationship between Pointers and Arrays (Cont.)

- The above 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 preceding notation is referred to as pointer/offset notation.
- The parentheses are necessary because the precedence of * is higher than the precedence of +.

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.

Relationship between Pointers and Arrays (Cont.)

- 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.

Relationship between Pointers and Arrays (Cont.)

- Example: fig07_20.c

```
 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 */
```

set **bPtr** to point to array b

array subscript 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 */
```

pointer/offset notation

Relationship between Pointers and Arrays (Cont.)

- Example: fig07_20.c

```
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 */
```

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

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
```

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.

Relationship between Pointers and Arrays (Cont.)

- Example: fig07_21.c

```
5  void copy1( char *s1, const char *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    return 0; /* indicates successful termination */
21 } /* end main */
```

declare four strings

Relationship between Pointers and Arrays (Cont.)

- Example: fig07_21.c

```
23 /* copy s2 to s1 using array notation */
24 void copy1( char *s1, const char *s2 )
25 {
26    int i; /* counter */
27
28    /* loop through strings */
29    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
30       ; /* do nothing in body */
31    } /* end for */
32 } /* end function copy1 */
```

use array subscription to copy the string

```
34 /* copy s2 to s1 using pointer notation */
35 void copy2( char *s1, const char *s2 )
36 {
37    /* loop through strings */
38    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
39       ; /* do nothing in body */
40    } /* end for */
41 } /* end function copy2 */
```

use pointer offset to copy the string

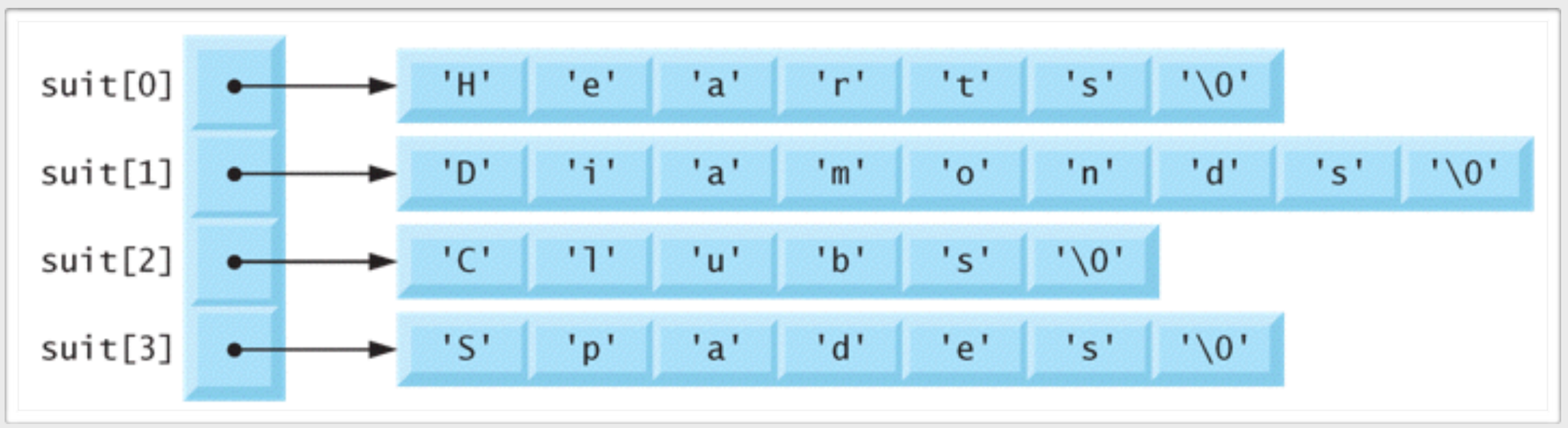
Arrays of Pointers

- 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.
- Consider the definition of string array suit, which might be useful in representing a deck of cards.

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

Arrays of Pointers (Cont.)

- 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.



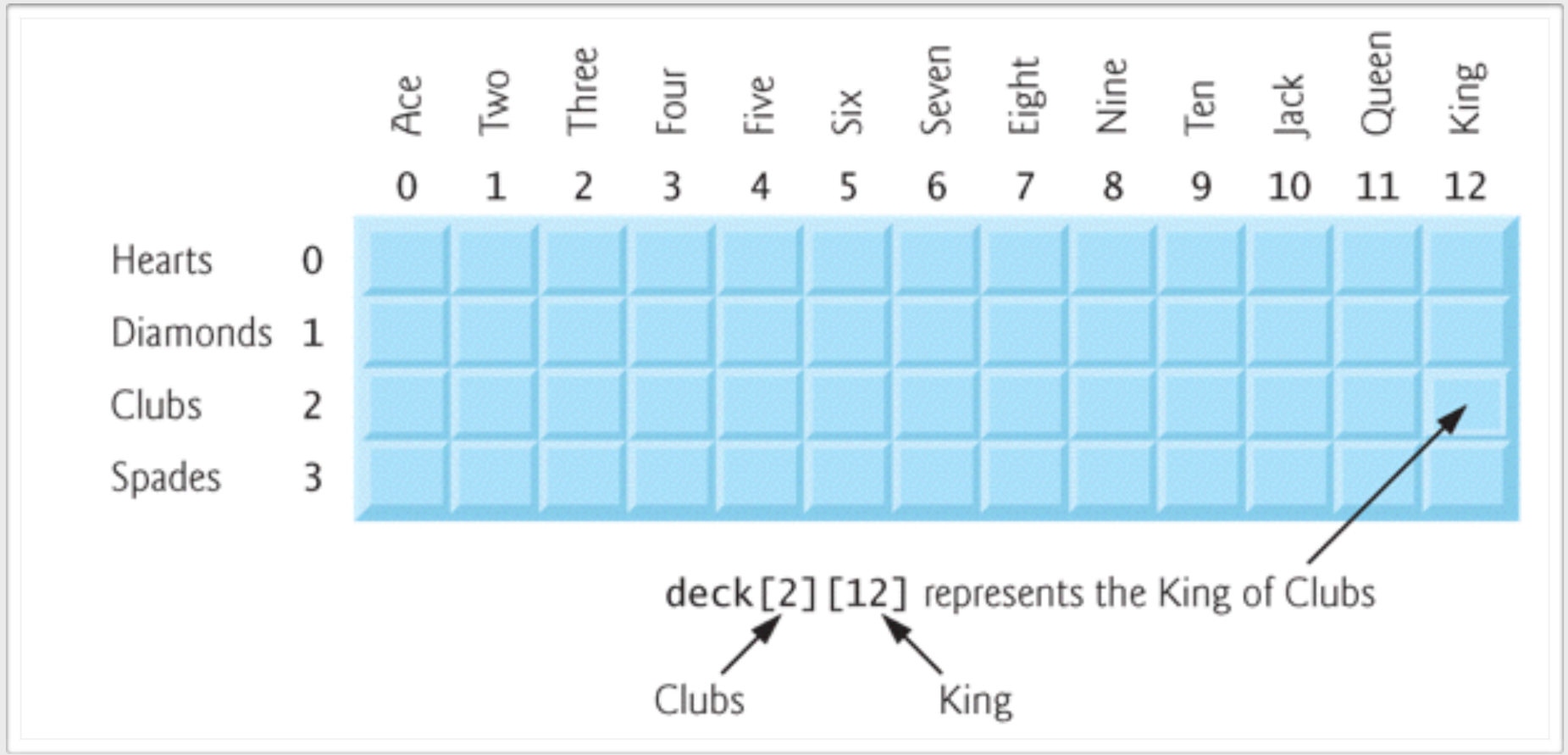
Arrays of Pointers (Cont.)

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

- 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.
- We use 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.

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



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

- Example: fig07_24.c

```
 8 void shuffle( int wDeck[][ 13 ] );
 9 void deal( const int wDeck[][ 13 ], const char *wFace[], const char *wSuit[] );
10
11 int main( void )
12 {
13    /* initialize suit array */
14    const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
15
16    /* initialize face array */
17    const char *face[ 13 ] =
18    { "Ace", "Deuce", "Three", "Four",
19       "Five", "Six", "Seven", "Eight",
20       "Nine", "Ten", "Jack", "Queen", "King" };
21
22    /* initialize deck array */
23    int deck[ 4 ][ 13 ] = { 0 };
24
25    srand( time( 0 ) ); /* seed random-number generator */
26
27    shuffle( deck ); /* shuffle the deck */
28    deal( deck, face, suit ); /* deal the deck */
29    return 0; /* indicates successful termination */
30 } /* end main */
```

four patterns

13 numbers

initialize a deck

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

- Example: fig07_24.c

```
33 void shuffle( int wDeck[][ 13 ] )
34 {
35    int row; /* row number */
36    int column; /* column number */
37    int card; /* counter */
38
39    /* for each of the 52 cards, choose slot of deck randomly */
40    for ( card = 1; card <= 52; card++ ) {
41
42       /* choose new random location until unoccupied slot found */
43       do {
44          row = rand() % 4;
45          column = rand() % 13;
46       } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
47
48       /* place card number in chosen slot of deck */
49       wDeck[ row ][ column ] = card;
50    } /* end for */
51 } /* end function shuffle */
```

shuffle the deck

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

- Example: fig07_24.c

```
54 void deal( const int wDeck[][ 13 ], const char *wFace[], const char *wSuit[] )
55 {
56    int card; /* card counter */
57    int row; /* row counter */
58    int column; /* column counter */
59
60    /* deal each of the 52 cards */
61    for ( card = 1; card <= 52; card++ ) {
62
63       /* loop through rows of wDeck */
64       for ( row = 0; row <= 3; row++ ) {
65
66          /* loop through columns of wDeck for current row */
67          for ( column = 0; column <= 12; column++ ) {
68
69             /* if slot contains current card, display card */
70             if ( wDeck[ row ][ column ] == card ) {
71                printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
72                   card % 2 == 0 ? '\n' : '\t' );
73             } /* end if */
74          } /* end for */
75       } /* end for */
76    } /* end for */
77 } /* end function deal */
```

search for each slot, and if slot contains current card, display card

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

- Example: fig07_24.c

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

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.
- You may try to correct this deficiency!

Pointers to Functions

- A pointer to a function contains the address of the function in memory.
- 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.

Pointers to Functions (Cont.)

- Example: fig07_26.c

```
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 );
```

function pointer

```
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
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 */
```

1 for ascending;
2 for descending

Pointers to Functions (Cont.)

- Example: fig07_26.c

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

function pointer

invoke the function via function pointer **`(*compare)`**

Pointers to Functions (Cont.)

- Example: fig07_26.c

```
79 void swap( int *element1Ptr, int *element2Ptr )
80 {
81    int hold; /* temporary holding variable */
82
83    hold = *element1Ptr;
84    *element1Ptr = *element2Ptr;
85    *element2Ptr = hold;
86 } /* end function swap */
```

```
90 int ascending( int a, int b )
91 {
92    return b < a; /* swap if b is less than a */
93
94 } /* end function ascending */
```

```
98 int descending( int a, int b )
99 {
100    return b > a; /* swap if b is greater than a */
101
102 } /* end function descending */
```

```
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
```

Pointers to Functions (Cont.)

- The following parameter appears in the function header for bubble

```
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.

Pointers to Functions (Cont.)

- 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.

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.
- The user's choice is used as a subscript in the array, and the pointer in the array is used to call the function.

Pointers to Functions (Cont.)

- Example: fig07_28.c

```c
/* 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 );

   /* 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 */
```

an array of three pointers to functions

invoke the corresponding function

Pointers to Functions (Cont.)

- Example: fig07_28.c

```c
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 */

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: 1
You entered 1 so function2 was called

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: 2
You entered 2 so function3 was called

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

Pointers to Functions (Cont.)

- The definition:

```
void (*f[3]) (int) = {function1, function2,
function3}
```

"**f** is an array of 3 pointers to functions that each take an int as an argument and return void." The array is initialized with the names of the three functions.

- In the function call,

```
(*f[choice]) (choice);
```

f[choice] selects the pointer at location choice in the array.

the pointer is dereferenced to call the function, and choice is passed as the argument to the function.

C Structures, Unions, Bit Manipulations and Enumerations

Objectives

- In this part, you'll learn
  - To create and use structures, unions and enumerations
  - To pass structures to functions by value and by reference
  - To manipulate data with the bitwise operations
  - To create bit field for storing data compactly

Introduction

- Structures—sometimes referred to as aggregates—are collections of related variables under one name.
- Structures may contain variables of many different data types—in contrast to arrays that contain only elements of the same data type.

Structure Definitions

- Structures are derived data types—they are constructed using objects of other types.
- Consider the following structure definition:

```
struct card {
   char *face;
   char *suit;
};
```

  - Keyword **`struct`** introduces the structure definition.
  - The identifier card is the structure tag, which names the structure definition and is used with the keyword **`struct`** to declare variables of the structure type.

Structure Definitions (Cont.)

- Variables declared within the braces of the structure definition are the structure's members.
- Each structure definition must end with a semicolon.
- The definition of **`struct card`** contains members **`face`** and **`suit`** of type **`char *`**.

Common Programming Error 10.1

Forgetting the semicolon that terminates a structure definition is a syntax error.

Structure Definitions (Cont.)

- Structure members can be of many types.
- For example

```
struct employee {
    char firstName[ 20 ];
    char lastName[ 20 ];
    int age;
    char gender;
    double hourlySalary;
};
```

Structure Definitions (Cont.)

- A structure cannot contain an instance of itself.
- For example,

```
struct employee2 {
    char firstName[ 20 ];
    char lastName[ 20 ];
    int age;
    char gender;
    double hourlySalary;
    struct employee2 person; /* ERROR */
    struct employee2 *ePtr; /* pointer */
};
```

Structure Definitions (Cont.)

- **`struct employee2 person`**
  - contains an instance of itself (person), which is an error.
- **`struct employee2 *ePtr;`**
  - A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure.
  - Self-referential structures are used to build linked data structures

Structure Definitions (Cont.)

- Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that is used to define variables.
- Structure variables are defined like variables of other types.
- The definition

```
struct card aCard, deck[ 52 ], *cardPtr;
```

Structure Definitions (Cont.)

- For example, the preceding definition could have been incorporated into the struct card structure definition as follows:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Structure Definitions (Cont.)

- The structure tag name is optional.

```
struct {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Structure Definitions (Cont.)

- The only valid operations that may be performed on structures are:
  - assigning structure variables to structure variables of the same type,
  - taking the address (**&**) of a structure variable,
  - accessing the members of a structure variable
  - using the **`sizeof`** operator to determine the size of a structure variable

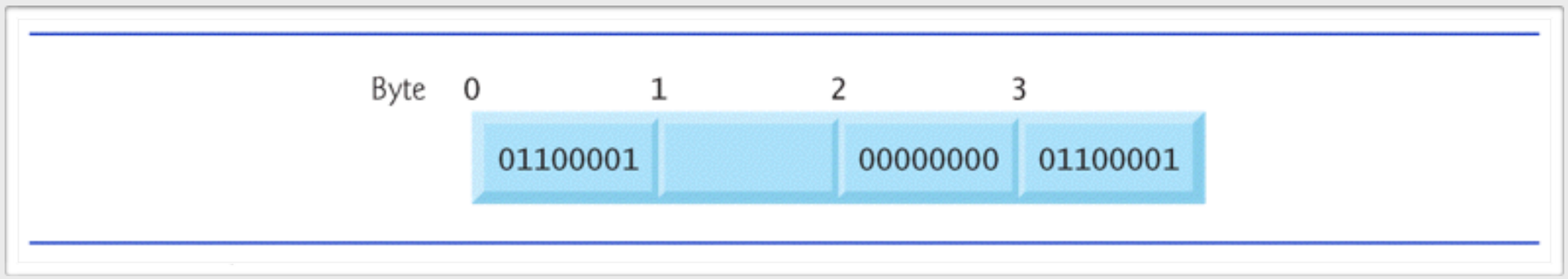
Structure Definitions (Cont.)

- Structures may not be compared using operators == and !=, because structure members are not necessarily stored in consecutive bytes of memory.

Structure Definitions (Cont.)

- Consider the following structure definition, in which **`sample1`** and **`sample2`** of type struct `example` are declared:

```
struct example {
    char c;
    int i;
} sample1, sample2;
```



Initializing Structures

- To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
- For example, the declaration

```
struct card {  // define a new data type
    char *face;
    char *suit;
};
```

```
struct card aCard = { "Three",
"Hearts" };
// declaration and initialization
```

Accessing Structure Members

- Two operators are used to access members of structures: the structure member operator (`.`)—also called the dot operator—and the structure pointer operator (`->`)—also called the arrow operator.
- For example, to print member **`suit`** of structure variable **`aCard`**, use the statement

```
printf( "%s", aCard.suit ); /*displays
Hearts */
```

Accessing Structure Members (Cont.)

- The structure pointer operator—consisting of a minus (-) sign and a greater than (>) sign with no intervening spaces—accesses a structure member via a pointer to the structure.
- To print member **suit** of structure **aCard** with pointer **cardPtr**, use the statement

```
printf( "%s", cardPtr->suit ); /*
displays Hearts */
```

Accessing Structure Members (Cont.)

- The expression **`cardPtr->suit`** is equivalent to **`(*cardPtr).suit`**, which dereferences the pointer and accesses the member **`suit`** using the structure member operator.
- The parentheses are needed here because the structure member operator (`.`) has a higher precedence than the pointer dereferencing operator (*).

Accessing Structure Members (Cont.)

- Example: fig10_02.c

```c
struct card {
   char *face; /* define pointer face */
   char *suit; /* define pointer suit */
}; /* end structure card */

int main( void ) {
   struct card aCard; /* define one struct card variable */
   struct card *cardPtr; /* define a pointer to a struct card */

   /* place strings into aCard */
   aCard.face = "Ace";
   aCard.suit = "Spades";

   cardPtr = &aCard; /* assign address of aCard to cardPtr */

   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
           cardPtr->face, " of ", cardPtr->suit,
           ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
   return 0; /* indicates successful termination */
} /* end main */
```

define a new data type named **`card`**

use the new type to declare two variables: **`aCard`** and **`cardPtr`**

use dot notation to access its members

use **`Ptr->member`** and **`(*Ptr).member`** to access its members

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Using Structures with Functions

- When structures or individual structure members are passed to a function, they are passed by value.
  - Therefore, the members of a caller's structure cannot be modified by the called function.
- To pass a structure by reference, pass the address of the structure variable.

Common Programming Error 10.6

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

`typedef`

- The keyword **`typedef`** provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter type names.
- For example, the statement

  ```
  typedef struct card Card;
  ```

  defines the new type name **`Card`** as a synonym for type **`struct card`**.

`typedef` (Cont.)

- For example, the following definition

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

creates the structure type **`Card`** without the need for a separate **`typedef`** statement.

`typedef` (Cont.)

- Creating a new name with **`typedef`** does not create a new type; **`typedef`** simply creates a new type name, which may be used as an alias for an existing type name.

Example: High-Performance Card Shuffling and Dealing Simulation

- Based on the card shuffling and dealing simulation discussed in Chapter 7.
- The program represents the deck of cards as an array of structures.
- The program uses high-performance shuffling and dealing algorithms.

Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- Example: fig10_03.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* card structure definition */
struct card {
   const char *face; /* define pointer face */
   const char *suit; /* define pointer suit */
}; /* end structure card */

typedef struct card Card; /* new type name for struct card */
```

card structure definition

new type name for **`struct card`**

Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- Example: fig10_03.c

```
20 int main( void ) {
21    Card deck[ 52 ]; /* define array of Cards */
22
23    /* initialize array of pointers */
24    const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
25       "Six", "Seven", "Eight", "Nine", "Ten",
26       "Jack", "Queen", "King"};
27
28    /* initialize array of pointers */
29    const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
30
31    srand( time( NULL ) ); /* randomize */
32
33    fillDeck( deck, face, suit ); /* load the deck with Cards */
34    shuffle( deck ); /* put Cards in random order */
35    deal( deck ); /* deal all 52 Cards */
36    return 0; /* indicates successful termination */
37 } /* end main */
```

define array of `Cards`

Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- Example: fig10_03.c

```
40 void fillDeck( Card * const wDeck, const char * wFace[],
41         const char * wSuit[] ) {
42    int i; /* counter */
43
44    /* loop through wDeck */
45    for ( i = 0; i <= 51; i++ ) {
46       wDeck[ i ].face = wFace[ i % 13 ];
47       wDeck[ i ].suit = wSuit[ i / 13 ];
48    } /* end for */
49 } /* end function fillDeck */
```

receive an array of **`Cards`**

loop through **`wDeck`**

Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- Example: fig10_03.c

```
52 void shuffle( Card * const wDeck ) {
53    int i; /* counter */
54    int j; /* variable to hold random value between 0 - 51 */
55    Card temp; /* define temporary structure for swapping Cards */
56
57    /* loop through wDeck randomly swapping Cards */
58    for ( i = 0; i <= 51; i++ ) {
59       j = rand() % 52;
60       temp = wDeck[ i ];
61       wDeck[ i ] = wDeck[ j ];
62       wDeck[ j ] = temp;
63    } /* end for */
64 } /* end function shuffle */
```

randomly swapping **`Cards`**; a total of 52 swaps are made in a single pass of the entire array

Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- Example: fig10_03.c

```
67 void deal( const Card * const wDeck ) {
68    int i; /* counter */
69
70    /* loop through wDeck */
71    for ( i = 0; i <= 51; i++ ) {
72       printf( "%5s of %-8s%s", wDeck[ i ].face, wDeck[ i ].suit,
73             ( i + 1 ) % 4 ? "  " : "\n" );
74    } /* end for */
75 } /* end function deal */
```

print the **`card face`** and **`suit`**

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

Unions

- A **`union`** is a derived data type—like a structure—with members that share the same storage space.
- For different situations in a program, some variables may not be relevant, but other variables are—so a union shares the space instead of wasting storage on variables that are not being used.
- The members of a union can be of any data type.

Unions (Cont.)

- A union is declared with keyword **union** in the same format as a structure.
- The union definition

```
union number {
    int x;
    double y;
};
```

  indicates that **number** is a **union** type with members **int x** and **double y**.

- The union definition is normally placed in a header and included in all source files that use the union type.

`Unions` (Cont.)

- The operations that can be performed on a union are the following:
  - assigning (=) a union to another union of the same type,
  - taking the address (**&**) of a union variable,
  - and accessing union members using the structure member operator and the structure pointer operator.
- Unions may not be compared using operators == and **!=** for the same reasons that structures cannot be compared.

`Unions` (Cont.)

- In a declaration, a union may be initialized with a value of the same type as the first union member.
- For example, with the preceding union, the declaration

```
union number value = { 10 };
```

is a valid initialization of union variable value.

Unions (Cont.)

- Example: fig10_05.c

```c
union number {
    int x;
    double y;
}; /* end union number */

int main( void ) {
    union number value; /* define union variable */

    value.x = 100; /* put an integer into the union */
    printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n\n\n",
            "Put a value in the integer member",
            "and print both members.",
            "int:", value.x,
            "double:", value.y );

    value.y = 100.0; /* put a double into the same union */
    printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
            "Put a value in the floating member",
            "and print both members.",
            "int:", value.x,
            "double:", value.y );
```

use union to define a new type named **number**

define an union variable named **value**

put an integer into the union

put a double into the union

Unions (Cont.)

- Example: fig10_05.c

```
28    printf( "\n\n%s\n%s\n  %p\n\n%s\n  %p\n",
29            "Output the addresses of the two members:",
30            "x:", &(value.x),
31            "y:", &(value.y));
```

two members are at the same address

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

double:
  0.000000
```

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

double:
  100.000000
```

```
Output the addresses of the two members:
x:
  0x7fff5fbfed90

y:
  0x7fff5fbfed90
```

Unions (Cont.)

- Example: fig10_06.c

```c
union udata {
    int x;
    long y;
    double z;
    char *a;
};

struct sdata {
    int x;
    long y;
    double z;
    char *a;
};

int main(void) {
    printf("%ld\n", sizeof(union udata));
    printf("%ld\n", sizeof(struct sdata));
    return 0;
}
```

define a union with four members

define a struct with the same four members

output the sizes of the above data types

```
8
32
```

Bitwise Operators

- Each bit can assume the value 0 or the value 1.
- On most systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type **`char`**.
- The bitwise operators are used to manipulate the bits of integral operands (**`char`**, **`short`**, **`int`** and **`long`**; both **`signed`** and **`unsigned`**).
- The bitwise operators are bitwise AND (**`&`**), bitwise inclusive OR (**`|`**), bitwise exclusive OR (**`^`**), left shift (**`<<`**), right shift (**`>>`**) and complement (**`~`**).

Bitwise Operators (Cont.)

Operator		Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
|	bitwise inclusive 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 the 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.

Bitwise Operators (Cont.)

- Example: fig10_07.c

```
5  void displayBits( unsigned value ); /* prototype */
6
7  int main( void ) {
8      unsigned x; /* variable to hold user input */
9
10     printf( "Enter an unsigned integer: " );
11     scanf( "%u", &x );
12
13     displayBits( x );
14     return 0; /* indicates successful termination */
15 } /* end main */
```

enter an unsigned integer

Bitwise Operators (Cont.)

- Example: fig10_07.c

```
18 void displayBits( unsigned value ) {
19    unsigned c; /* counter */
20
21    /* define displayMask and left shift 31 bits */
22    unsigned displayMask = 1 << 31;
23
24    printf( "%10u = ", value );
25
26    /* loop through bits */
27    for ( c = 1; c <= 32; c++ ) {
28       putchar( value & displayMask ? '1' : '0' );
29       value <<= 1; /* shift value left by 1 */
30
31       if ( c % 8 == 0 ) { /* output space after 8 bits */
32          putchar( ' ' );
33       } /* end if */
34    } /* end for */
35
36    putchar( '\n' );
37 } /* end function displayBits */
```

define a mask
10000000 00000000
00000000 00000000

all the bits except the high-order bit in **`value`** are "masked off", because any bit "ANDed" with 0 yields 0

Bitwise Operators (Cont.)

- Often, the bitwise AND operator is used with an operand called a mask—an integer value with specific bits set to 1.
- Masks are used to hide some bits in a value while selecting other bits.
- When **`value`** and **`displayMask`** are combined using **`&`**, all the bits except the high-order bit in variable **`value`** are "masked off" (hidden), because any bit "ANDed" with 0 yields 0.

Bitwise Operators (Cont.)

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Bitwise Operators (Cont.)

- Example: fig10_09.c

```
 9     unsigned number1; /* define number1 */
10     unsigned number2; /* define number2 */
11     unsigned mask; /* define mask */
12     unsigned setBits; /* define setBits */
13
14     /* demonstrate bitwise AND (&) */
15     number1 = 65535;
16     mask = 1;
17     printf( "The result of combining the following\n" );
18     displayBits( number1 );
19     displayBits( mask );
20     printf( "using the bitwise AND operator & is\n" );
21     displayBits( number1 & mask );
```

demonstrate the use of the bitwise AND operator

```
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
```

Bitwise Operators (Cont.)

- Example: fig10_09.c

```
23    /* demonstrate bitwise inclusive OR (|) */
24    number1 = 15;
25    setBits = 241;
26    printf( "\nThe result of combining the following\n" );
27    displayBits( number1 );
28    displayBits( setBits );
29    printf( "using the bitwise inclusive OR operator | is\n" );
30    displayBits( number1 | setBits );
31
32    /* demonstrate bitwise exclusive OR (^) */
33    number1 = 139;
34    number2 = 199;
35    printf( "\nThe result of combining the following\n" );
36    displayBits( number1 );
37    displayBits( number2 );
38    printf( "using the bitwise exclusive OR operator ^ is\n" );
39    displayBits( number1 ^ number2 );
```

demonstrate the use of the bitwise OR operator

demonstrate the use of the bitwise XOR operator

```
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
```

Bitwise Operators (Cont.)

- Example: fig10_09.c

```
41    /* demonstrate bitwise complement (~) */
42    number1 = 21845;
43    printf( "\nThe one's complement of\n" );
44    displayBits( number1 );
45    printf( "is\n" );
46    displayBits( ~number1 );
```

demonstrate the use of the bitwise NOT operator

```
The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Bitwise Operators (Cont.)

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 10.11 | Results of combining two bits with the bitwise inclusive OR operator |.

Bitwise Operators (Cont.)

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 10.12 | Results of combining two bits with the bitwise exclusive OR operator ^.

Bitwise Operators (Cont.)

- Example: fig10_13.c

```
 8    unsigned number1 = 960; /* initialize number1 */
 9
10    /* demonstrate bitwise left shift */
11    printf( "\nThe result of left shifting\n" );
12    displayBits( number1 );
13    printf( "8 bit positions using the " );
14    printf( "left shift operator << is\n" );
15    displayBits( number1 << 8 );
16
17    /* demonstrate bitwise right shift */
18    printf( "\nThe result of right shifting\n" );
19    displayBits( number1 );
20    printf( "8 bit positions using the " );
21    printf( "right shift operator >> is\n" );
22    displayBits( number1 >> 8 );
```

left shift 8 bits

right shift 8 bits

```
The result of left shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
 245760 = 00000000 00000011 11000000 00000000

The result of right shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000000 00000000 00000011
```

Bitwise Operators (Cont.)

- The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- Bits vacated to the right are replaced with **0**s; 1s shifted off the left are lost.
- The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand.
- Performing a right shift on an **`unsigned`** integer causes the vacated bits at the left to be replaced by **0**s; 1s shifted off the right are lost.

Bitwise Operators (Cont.)

Common Programming Error 10.12

The result of shifting a value is undefined if the right operand is negative or if the right operand is larger than the number of bits in which the left operand is stored.

Bitwise Operators (Cont.)

Bitwise assignment operators	
&=	Bitwise AND assignment operator.
|=	Bitwise inclusive OR assignment operator.
^=	Bitwise exclusive OR assignment operator.
<<=	Left-shift assignment operator.
>>=	Right-shift assignment operator.

Fig. 10.14 | The bitwise assignment operators.

Bit Fields

- C enables you to specify the number of bits in which an **`unsigned`** or **`int`** member of a structure or union is stored.
- This is referred to as a bit field.
- Bit fields enable better memory utilization by storing data in the minimum number of bits required.
- Bit field members must be declared as **`int`** or **`unsigned`**.

Bit Fields (Cont.)

- Consider the following structure definition:

```
struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

which contains three **`unsigned`** bit fields—**`face`**, **`suit`** and **`color`**—used to represent a card from a deck of 52 cards.

Bit Fields (Cont.)

- A bit field is declared by following an **`unsigned`** or **`int`** member name with a colon (**`:`**) and an integer constant representing the width of the field (i.e., the number of bits in which the member is stored).
- The preceding structure definition indicates that member **`face`** is stored in 4 bits, member **`suit`** is stored in 2 bits and member **`color`** is stored in 1 bit.

Bit Fields (Cont.)

- Example: fig10_16.c

```
 7 struct bitCard {
 8    unsigned face : 4; /* 4 bits; 0-15 */
 9    unsigned suit : 2; /* 2 bits; 0-3 */
10    unsigned color : 1; /* 1 bit; 0-1 */
11 }; /* end struct bitCard */
12
13 typedef struct bitCard Card; /* new type name for struct bitCard */
14
15 void fillDeck( Card * const wDeck ); /* prototype */
16 void deal( const Card * const wDeck ); /* prototype */
17
18 int main( void ) {
19    Card deck[ 52 ]; /* create array of Cards */
20
21    fillDeck( deck );
22    deal( deck );
23    return 0; /* indicates successful termination */
24 } /* end main */
```

define a struct and use bit fields

a new type name for struct **`bitCard`**

create a deck of Cards

Bit Fields (Cont.)

- Example: fig10_16.c

```
27 void fillDeck( Card * const wDeck ) {
28    int i; /* counter */
29
30    /* loop through wDeck */
31    for ( i = 0; i <= 51; i++ ) {
32       wDeck[ i ].face = i % 13;
33       wDeck[ i ].suit = i / 13;
34       wDeck[ i ].color = i / 26;
35    } /* end for */
36 } /* end function fillDeck */
```

initialize the deck by looping through **`wDeck`**

Bit Fields (Cont.)

- Example: fig10_16.c

```
40 void deal( const Card * const wDeck ) {
41    int k1; /* subscripts 0-25 */
42    int k2; /* subscripts 26-51 */
43
44    /* loop through wDeck */
45    for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
46       printf( "Card:%3d  Suit:%2d  Color:%2d   ",
47               wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
48       printf( "Card:%3d  Suit:%2d  Color:%2d\n",
49               wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
50    } /* end for */
51 } /* end function deal */
```

print out the deck

```
Card:  0  Suit: 0  Color: 0   Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0   Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0   Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0   Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0   Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0   Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0   Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0   Card:  7  Suit: 2  Color: 1
```

Enumeration Constants

- C provides one final user-defined type called an enumeration.
- An enumeration, introduced by the keyword enum, is a set of integer enumeration constants represented by identifiers.
- For example, the enumeration

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

creates a new type, **`enum months`**, in which the identifiers are set to the integers **`0`** to **`11`**, respectively.

Enumeration Constants (Cont.)

- Values in an **`enum`** start with **`0`**, unless specified otherwise, and are incremented by **`1`**.
- To number the months 1 to 12, use the following enumeration:

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

- The identifiers in an enumeration must be unique.

Enumeration Constants (Cont.)

- Example: fig10_18.c

```
6  enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
7
8  int main( void ) {
9     enum months month; /* can contain any of the 12 months */
10
11    /* initialize array of pointers */
12    const char *monthName[] = { "", "January", "February", "March",
13       "April", "May", "June", "July", "August", "September", "October",
14       "November", "December" };
15
16    /* loop through months */
17    for ( month = JAN; month <= DEC; month++ ) {
18       printf( "%2d%11s\n", month, monthName[ month ] );
19    } /* end for */
20
21    return 0; /* indicates successful termination */
22 } /* end main */
```

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
```

create enumeration constants

`month` contain any of the 12 months

loop through months

C Preprocessor

Objectives

- In this part, you'll learn
  - To use **`#include`** to develop large programs
  - To use **`#define`** to create macros and macros with arguments
  - To use conditional compilation
  - To display error messages during conditional compilation
  - To use assertion to test if the values of expressions are correct

Introduction

- The C preprocessor executes before a program is compiled.
- Some actions can be defined in preprocessor
  - symbolic constants and macros
  - conditional compilation of program
  - conditional execution of preprocessor directives
- Preprocessor directives begin with #

`#include` Preprocessor Directive

- The `#include` directive causes a copy of a specified file to be included in place of the directive.
- The two forms of the `#include` directive are:

```
#include <filename>
#include "filename"
```

- If the file name is enclosed in quotes, the preprocessor starts searches in the same directory as the file being compiled.

`#include` Preprocessor Directive (Cont.)

- If the file name is enclosed in angle brackets (< and >)—used for standard library headers—the search is performed in an implementation-dependent manner, normally through predesignated compiler and system directories (e.g., /usr/bin)

`#include` Preprocessor Directive (Cont.)

- A header containing declarations common to the separate program files is often created and included in the file.
- Examples of such declarations are structure and union declarations, enumerations and function prototypes.

`#include` Preprocessor Directive Symbolic Constants

- The `#define` directive creates symbolic constants—constants represented as symbols—and macros—operations defined as symbols.
- The `#define` directive format is

```
#define  identifier  replacement-text
```

- For example
  - `#define PI  3.14159`
  - ❗ ~~`#define PI = 3.14159`~~

#include Preprocessor Directive Symbolic Constants (Cont.)

Good Programming Practice 13.1

Using meaningful names for symbolic constants helps make programs more self-documenting.

Good Programming Practice 13.2

By convention, symbolic constants are defined using only uppercase letters and underscores.

`#define` Preprocessor Directive: Macros

- A macro is an identifier defined in a **`#define`** preprocessor directive.
- As with symbolic constants, the macro-identifier is replaced in the program with the replacement-text before the program is compiled.
- In a macro with arguments, the arguments are substituted in the replacement text, then the macro is expanded.

`#define` Preprocessor Directive: Macros (Cont.)

- Consider the following macro definition with one argument for the area of a circlee

```
#define CIRCLE_AREA( x ) ( ( PI ) *
( x ) * ( x ) )
```

- For example, the statement

```
area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( ( 3.14159 ) * ( 4 ) * ( 4 ) );
```

`#define` Preprocessor Directive: Macros (Cont.)

- The parentheses around each **`x`** in the replacement text force the proper order of evaluation when the macro argument is an expression.
- For example, the statement

```
area = CIRCLE_AREA( c + 2 );
```

is expanded to

```
area = (( 3.14159 ) * ( c + 2 ) * ( c + 2 ));
```

which evaluates correctly because the parentheses force the proper order of evaluation.

`#define` Preprocessor Directive: Macros (Cont.)

Common Programming Error 13.1

Forgetting to enclose macro arguments in parentheses in the replacement text can lead to logic errors.

`#define` Preprocessor Directive: Macros (Cont.)

- Macro **`CIRCLE_AREA`** could be defined as a function.
- The advantages of macro **`CIRCLE_AREA`** are that macros insert code directly in the program—avoiding function call overhead.

`#define` Preprocessor Directive: Macros (Cont.)

Performance Tip 13.1

Macros can sometimes be used to replace a function call with inline code to eliminate the overhead of a function call. Today's optimizing compilers often inline functions for you, so many programmers no longer use macros for this purpose. C99 also provides the `inline` *keyword (see Appendix G).*

`#define` Preprocessor Directive: Macros (Cont.)

- The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
```

- For example, the statement

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

is expanded to

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

`#define` Preprocessor Directive: Macros (Cont.)

- Symbolic constants and macros can be discarded by using the **`#undef`** preprocessor directive.
- Directive **`#undef`** "undefines" a symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definition until it is undefined with **`#undef`**, or until the end of the file.

`#define` Preprocessor Directive: Macros (Cont.)

- A macro commonly defined in the **`stdio.h`** header is

```
#define getchar() getc( stdin )
```

- The macro definition of **`getchar`** uses function **`getc`** to get one character from the standard input stream.
- Expressions with side effects (i.e., variable values are modified) should not be passed to a macro because macro arguments may be evaluated more than once.

Conditional Compilation

- Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code.
- The conditional preprocessor construct is much like the if selection statement.
- Each of the conditional preprocessor directives evaluates a constant integer expression.

Conditional Compilation (Cont.)

- Consider the following preprocessor code:

```
#if !defined(MY_CONSTANT)
    #define MY_CONSTANT 0
#endif
```

- These directives determine if **MY_CONSTANT** is defined.

Conditional Compilation (Cont.)

- Every **`#if`** construct ends with **`#endif`**.
- Directives **`#ifdef`** and **`#ifndef`** are shorthand for **`#if defined`**(name) and **`#if !defined`**(name).
- A multiple-part conditional preprocessor construct may be tested by using the **`#elif`** (the equivalent of **`else if`** in an **`if`** statement) and the **`#else`** (the equivalent of **`else`** in an **`if`** statement) directives.
- These directives are frequently used to prevent header files from being included multiple times in the same source file.

Conditional Compilation (Cont.)

- During program development, it is often helpful to "comment out" portions of code to prevent it from being compiled.
- You can use the following preprocessor construct:

```
#if 0
   code prevented from compiling
#endif
```

- To enable the code to be compiled, replace the `0` in the preceding construct with `1`.

Conditional Compilation (Cont.)

- Conditional compilation is commonly used as a debugging aid.
- For example,

```
#ifdef DEBUG
   printf( "Variable x = %d\n", x );
#endif
```

causes a **`printf`** statement to be compiled in the program if the symbolic constant **`DEBUG`** has been defined (**`#define DEBUG`**) before directive **`#ifdef DEBUG`**.

Conditional Compilation (Cont.)

- When debugging is completed, the **`#define`** directive is removed from the source file (or commented out) and the **`printf`** statements inserted for debugging purposes are ignored during compilation.
- In larger programs, it may be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

and ## Operators

- The # operator causes a replacement text token to be converted to a string surrounded by quotes.
- Consider the following macro definition:

```
#define HELLO(x) printf( "Hello, " #x "\n" );
```

- When **HELLO(John)** appears in a program file, it is expanded to

```
printf( "Hello, " "John" "\n" );
```

- The string "John" replaces **#x** in the replacement text.
- Strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to

```
printf( "Hello, John\n" );
```

and ## Operators (Cont.)

- The ## operator concatenates two tokens.
- Consider the following macro definition:

```
#define TOKENCONCAT(x, y)  x ## y
```

  - When **TOKENCONCAT** appears in the program, its arguments are concatenated and used to replace the macro.
  - For example, **TOKENCONCAT(O, K)** is replaced by **OK** in the program.

Line Numbers

- The #line preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value.
- The directive

```
#line 100
```

starts line numbering from `100` beginning with the next source code line.

Line Numbers (Cont.)

- A file name can be included in the **`#line`** directive.
- The directive

  `#line 100 "file1.c"`

  indicates that lines are numbered from `100` beginning with the next source code line and that the name of the file for the purpose of any compiler messages is "**`file1.c`**".

- The directive normally is used to help make the messages produced by syntax errors and compiler warnings more meaningful.

Predefined Symbolic Constants

- Standard C provides predefined symbolic constants
- The identifiers for each of the predefined symbolic constants begin and end with *two underscores*.
- These identifiers and the **`defined`** identifier cannot be used in **`#define`** or **`#undef`** directives.

Predefined Symbolic Constants (Cont.)

Symbolic constant	Explanation
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file was compiled (a string of the form `"Mmm dd yyyy"` such as `"Jan 19 2002"`).
`__TIME__`	The time the source file was compiled (a string literal of the form `"hh:mm:ss"`).
`__STDC__`	The value 1 if the compiler supports Standard C.

Assertions

- The **`assert`** macro—defined in the **`<assert.h>`** header—tests the value of an expression.
- If the value of the expression is false (**`0`**), **`assert`** prints an error message and calls function **`abort`** (of the general utilities library—**`<stdlib.h>`**) to terminate program execution.
- This is a useful debugging tool for testing if a variable has a correct value.

Assertions (Cont.)

- For example, suppose variable **x** should never be larger than 10 in a program.
- An assertion may be used to test the value of **x** and print an error message if the value of **x** is incorrect.
- The statement would be

```
assert( x <= 10 );
```

- If **x** is greater than 10 when the preceding statement is encountered in a program, an error message containing the line number and file name is printed and the program terminates.

Assertions (Cont.)

- If the symbolic constant **`NDEBUG`** is defined, subsequent assertions will be ignored.
- Thus, when assertions are no longer needed, the line

```
#define NDEBUG
```

is inserted in the program file rather than deleting each assertion manually.