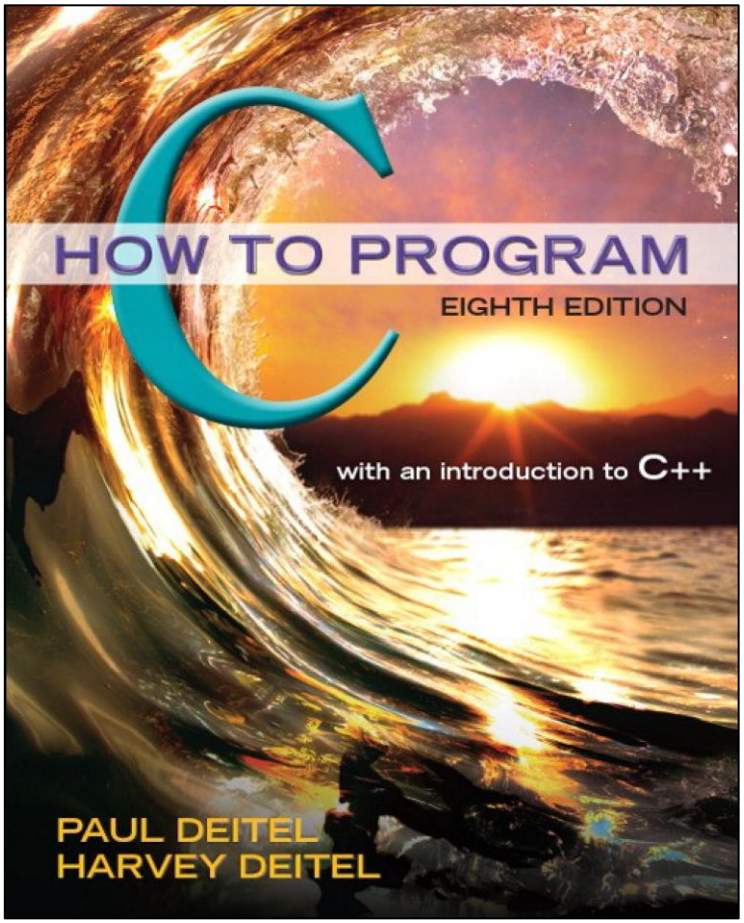
C How to Program

Eighth Edition



Chapter 7

C Pointers

Learning Objectives (1 of 2)

- Use pointers and pointer operators.
- Pass arguments to functions by reference using pointers.
- Understand the various placements of the `const` qualifier and how they affect what you can do with a variable.
- Use the `sizeof` operator with variables and types.
- Use pointer arithmetic to process the elements in arrays.
- Understand the close relationships among pointers. arrays and strings.

Learning Objectives (2 of 2)

- Define and use arrays of strings.
- Use pointers to functions.

Outline (1 of 4)

Outline (2 of 4)

Outline (3 of 4)

Outline (4 of 4)

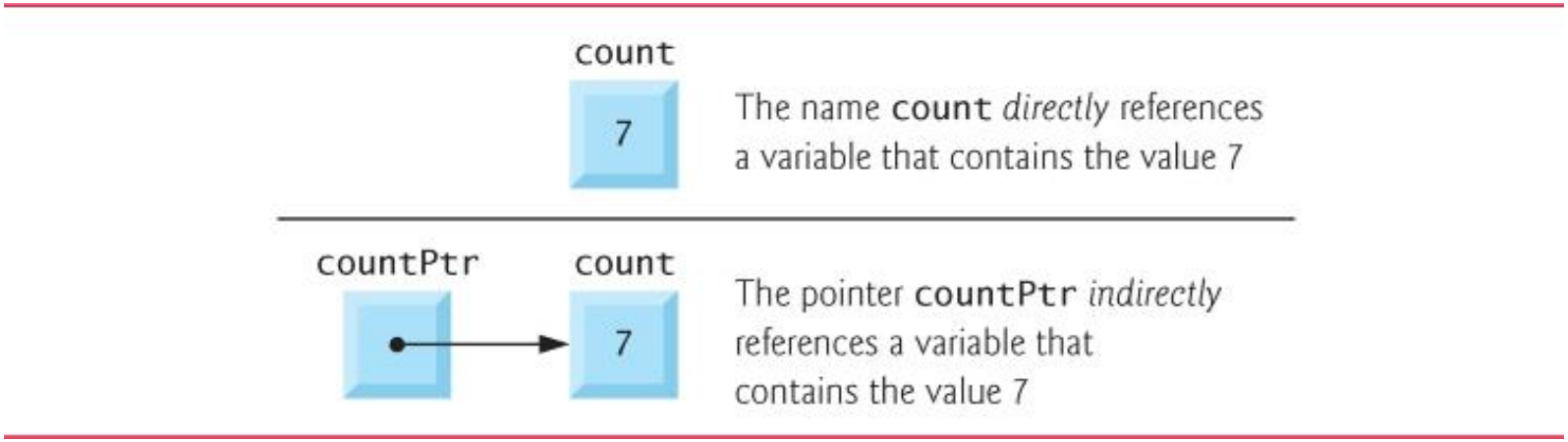
7.1 Introduction

- In this chapter, we discuss one of the most powerful features of the C programming language, the **pointer.**
- Pointers enable programs to simulate pass-by-reference, to pass functions between functions, 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.
- Chapter 10 examines the use of pointers with structures.
- Chapter 12 introduces dynamic memory management techniques and presents examples of creating and using dynamic data structures.

7.2 Pointer Variable Definitions and Initialization (1 of 5)

- 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.
- In this sense, a variable name **directly** references a value, and a pointer **indirectly** references a value (Figure 7.1).
- Referencing a value through a pointer is called **indirection.**

Figure 7.1 Directly and Indirectly Referencing a Variable



7.2 Pointer Variable Definitions and Initialization (2 of 5)

Declaring Pointers

- Pointers, like all variables, must be defined before they can be used.
- The definition
  - `int *countPtr, count;`

  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer) and is read (right to left), "`countPtr` is a pointer to `int`" or "`countPtr` points to an object of type `int`."
- Also, the variable `count` is defined to be an `int`, not a pointer to an `int`.

7.2 Pointer Variable Definitions and Initialization (3 of 5)

- The * applies **only** to `countPtr` in the definition.
- When * is used in this manner in a definition, it indicates that the variable being defined is a pointer.
- Pointers can be defined to point to objects of any type.
- To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown above, you should always declare only one variable per declaration.

7.2 Pointer Variable Definitions and Initialization (4 of 5)

Initializing and Assigning Values to Pointers

- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` points to **nothing**.
- `NULL` is a **symbolic constant** defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`).

7.2 Pointer Variable Definitions and Initialization (5 of 5)

- Initializing a pointer to `0` is equivalent to initializing a pointer to `Null`, but `Null` is preferred.
- When `0` is assigned, it's first converted to a pointer of the appropriate type.
- The value `0` is the **only** integer value that can be assigned directly to a pointer variable.

7.3 Pointer Operators (1 of 5)

- 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;
    ```

  the statement
  - ```
    yPtr = &y;
    ```

  assigns the **address** of the variable `y` to pointer variable `yPtr`.
- Variable `yPtr` is then said to "point to" `y`.
- Figure 7.2 (see slide 20) shows a schematic representation of memory after the preceding assignment is executed.

7.3 Pointer Operators (2 of 5)

Pointer Representation in Memory

- Figure 7.3 shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location `600000`, and pointer variable `yPtr` is stored at location `500000`.
- The operand of the address operator must be a variable; the address operator **cannot** be applied to constants or expressions.

Figure 7.2 Graphical Representation of a Pointer Pointing to an Integer Variable in Memory

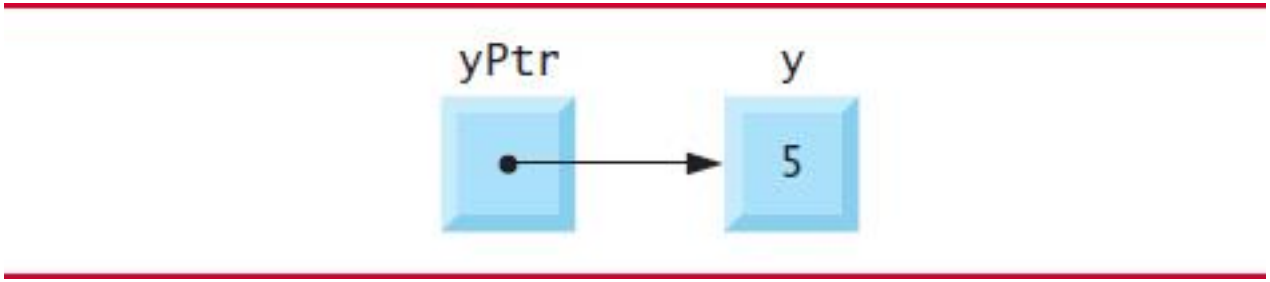


Figure 7.3 Representation of y and yPtr in Memory

7.3 Pointer Operators (3 of 5)

The Indirection (*) Operator

- 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**.

7.3 Pointer Operators (4 of 5)

Demonstrating the & and * Operators

- Figure 7.4 demonstrates the pointer operators `&` and `*`.
- The `printf` conversion specifier `%p` outputs the memory location as a **hexadecimal** integer on most platforms.
- (See Appendix C, for more information on hexadecimal integers.)
- Notice that the **address** of `a` and the **value** of `aPtr` are identical in the output, thus confirming that the address of `a` is indeed assigned to the pointer variable `aPtr`

7.3 Pointer Operators (5 of 5)

- The & and * operators are complements of one another—when they're both applied consecutively to `aPtr` in either order, the same result is printed.
- Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

Figure 7.4 Using the & and * Pointer Operators (1 of 2)

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main(void)
{
   int a = 7;
   int *aPtr = &a; // set aPtr to the address of a

   printf("The address of a is %p"
          "\nThe value of aPtr is %p", &a, aPtr);

   printf("\n\nThe value of a is %d"
          "\nThe value of *aPtr is %d", a, *aPtr);

   printf("\n\nShowing that * and & are complements of "
          "each other\n&*aPtr = %p"
          "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
}
```

Figure 7.4 Using the & and * Pointer Operators (2 of 2)

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

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

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Figure 7.5 Precedence and Associativity of the Operators Discussed So Far

Operators	Associativity	Type
[] () ++ (**postfix**) -- (**postfix**)	left to right	postfix
+ - ++ -- ! * & (**type**)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > s >=	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

7.4 Passing Arguments to Functions by Reference (1 of 6)

- There are two ways to pass arguments to a function—**pass-by-value** and **pass-by-reference.**
- **All arguments in C are passed by value.**
- Many functions require the capability to **modify variables in the caller** or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).
- In C, you use pointers and the indirection operator to **simulate** pass-by-reference.

7.4 Passing Arguments to Functions by Reference (2 of 6)

- When calling a function with arguments that should be modified, the **addresses** of the arguments are passed.
- This is normally accomplished by applying the address operator (`&`) to the variable (in the caller) whose value will be modified.
- As we saw in Chapter 6, arrays are **not** passed using operator `&` because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`).
- 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.

7.4 Passing Arguments to Functions by Reference (3 of 6)

Pass-By-Value

- The programs in Figure 7.6 and Figure 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`.
- Figure 7.6 passes the variable `number` by value to function `cubeByValue`
- The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement.
- The new value is assigned to `number` in `main`

Figure 7.6 Cube a Variable Using Pass-By-Value (1 of 2)

```
1  // Fig. 7.6: fig07_06.c
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
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 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22    return n * n * n; // cube local variable n and return result
23 }
```

Figure 7.6 Cube a Variable Using Pass-By-Value (2 of 2)

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

7.4 Passing Arguments to Functions by Reference (4 of 6)

Pass-By-Reference

- Figure 7.7 passes the variable `number` by reference—the address of `number` is passed—to function `cubeByReference`.
- Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr`.
- The function **dereferences** the pointer and cubes the value to which `nPtr` points, then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`.
- Figure 7.8 and Figure 7.9 analyze graphically and step-by-step the programs in Figure 7.6 and Figure 7.7, respectively.

Figure 7.7 Cube a Variable Using Pass-By-Reference with a Pointer Argument (1 of 2)

```
1  // Fig. 7.7: fig07_07.c
2  // Cube a variable using pass-by-reference with a pointer argument.
3
4  #include <stdio.h>
5
6  void cubeByReference(int *nPtr); // function 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 }
19
20 // calculate cube of *nPtr; actually modifies number in main
21 void cubeByReference(int *nPtr)
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 }
```

Figure 7.7 Cube a Variable Using Pass-By-Reference with a Pointer Argument (2 of 2)

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

7.4 Passing Arguments to Functions by Reference (5 of 6)

- A function receiving an **address** as an argument must define a **pointer parameter** to receive the address.
- For example, in Figure 7.7 the header for function `cubeByReference` is:
  - `void cubeByReference(int *nPtr)`
- The header specifies that `cubeByReference` **receives** the **address** of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.
- The function prototype for `cubeByReference` contains `int *` in parentheses.
- Names included for documentation purposes are ignored by the C compiler.

7.4 Passing Arguments to Functions by Reference (6 of 6)

- For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function `cubeByReference`.
- The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array.
- This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.
- The two forms are interchangeable.

Figure 7.8 Analysis of a Typical Pass-By-Value (1 of 3)

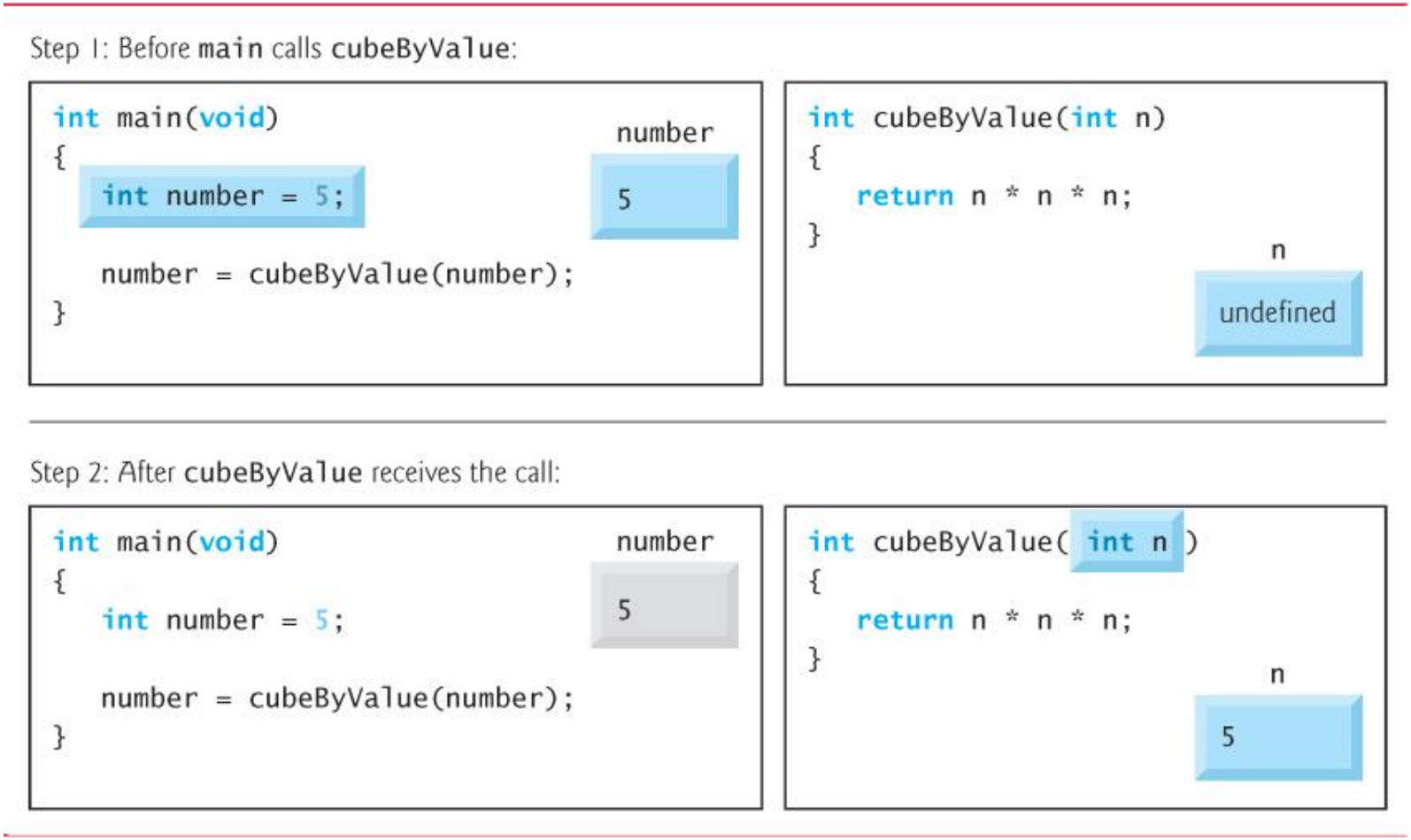
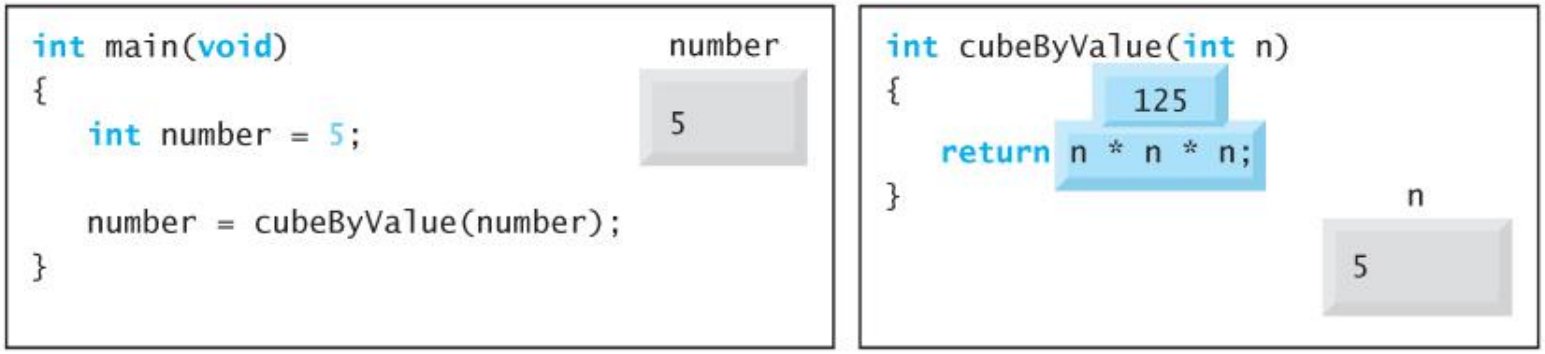


Figure 7.8 Analysis of a Typical Pass-By-Value (2 of 3)

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



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

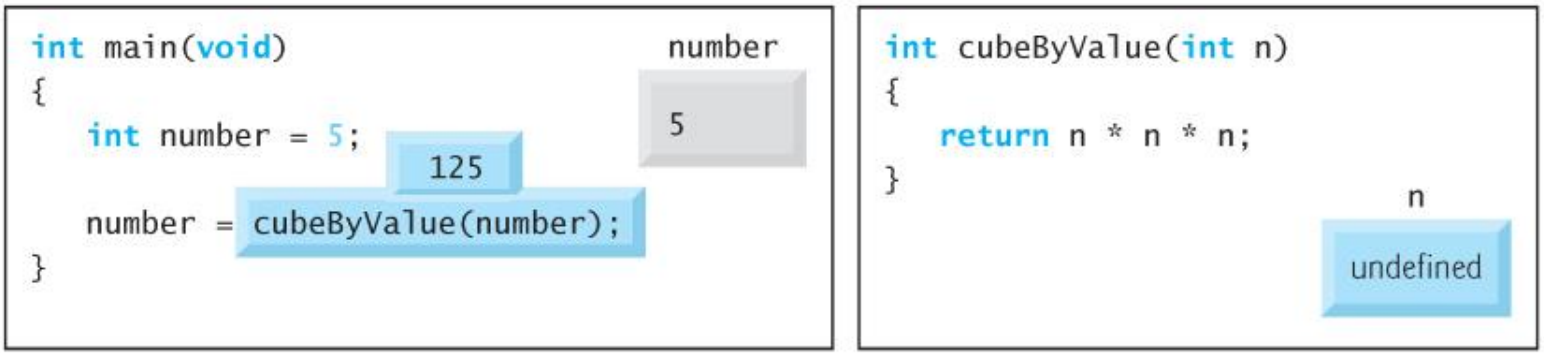


Figure 7.8 Analysis of a Typical Pass-By-Value (3 of 3)

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

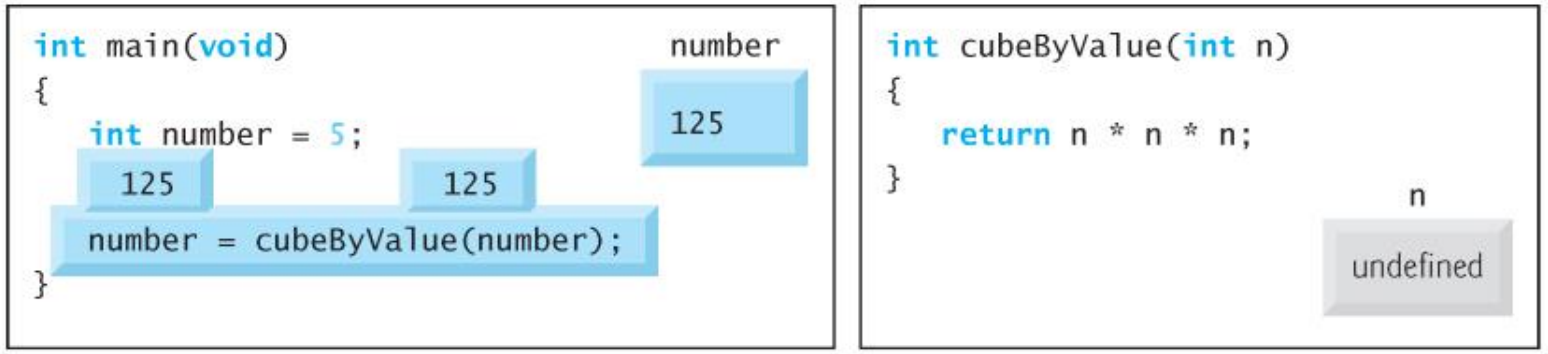


Figure 7.9 Analysis of a Typical Pass-By-Reference with a Pointer Argument (1 of 2)

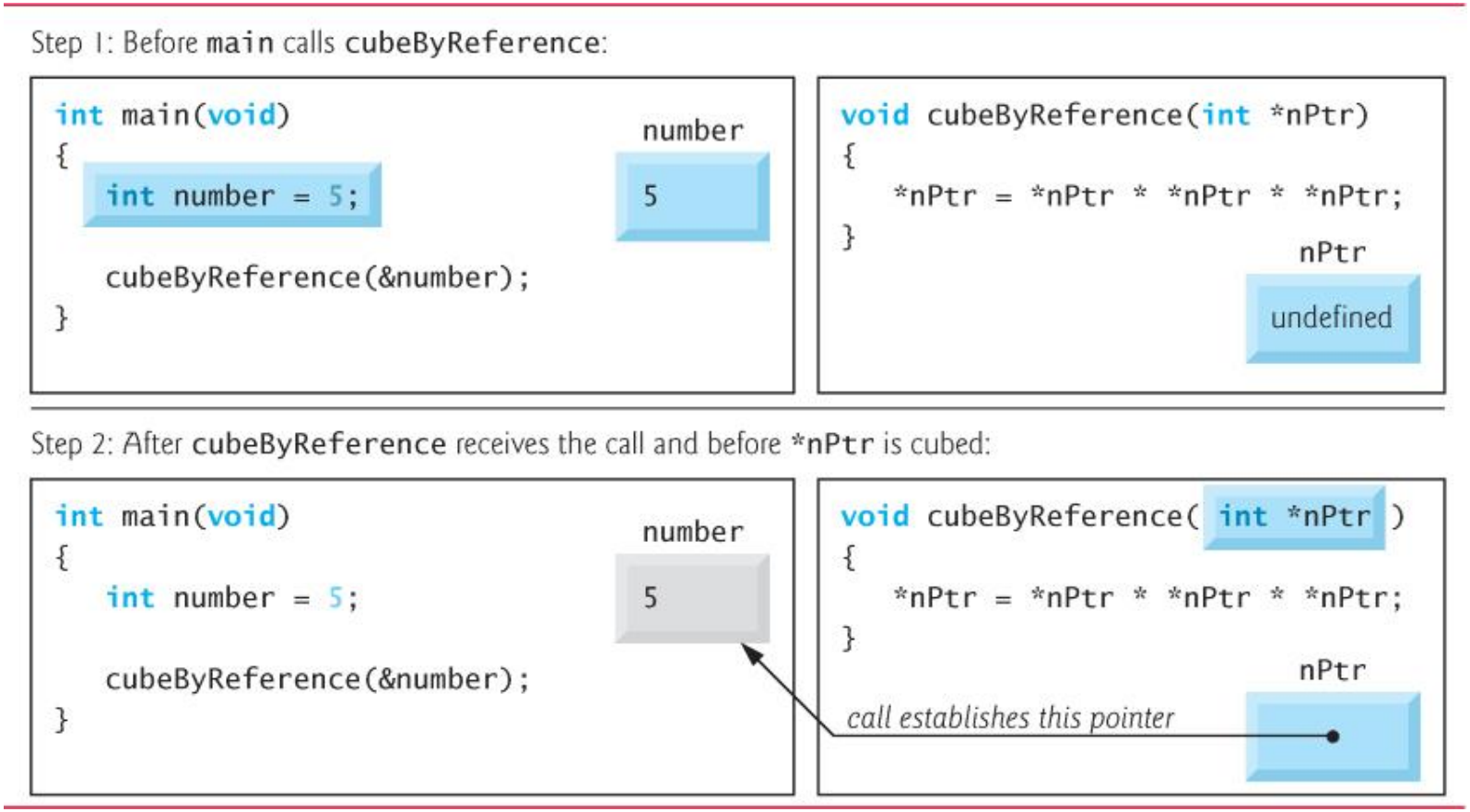
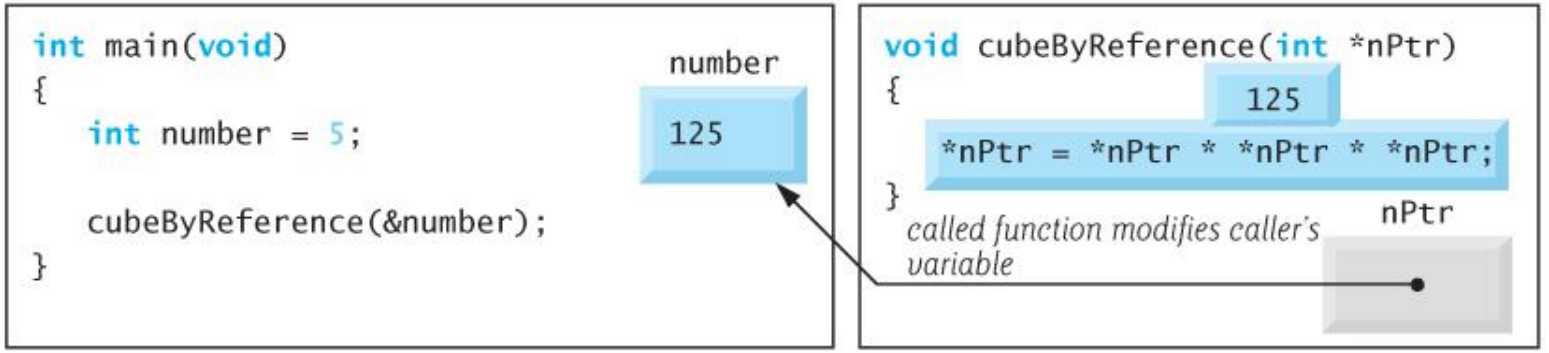


Figure 7.9 Analysis of a Typical Pass-By-Reference with a Pointer Argument (2 of 2)

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



7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data (1 of 2)

- 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`.
- Such a pointer might be used to receive a string as an argument to a function that processes (and possibly modifies) each character in the string.

7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data (2 of 2)

- Function `convertToUppercase` of Figure 7.10 declares its parameter, a non-constant pointer to **non-constant data** called `sPtr (char *sPtr)`
- The function processes the array `string` (pointed to by `sPtr`) one character at a time.
- C standard library function `touper` from the `<ctype.h>` header is called to convert each character to its corresponding uppercase letter—if the original character is not a letter or is already uppercase, `toupper` returns the original character.
- Line 23 moves the pointer to the next character in the string.

Figure 7.10 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data (1 of 2)

```
1  // Fig. 7.10: fig07_10.c
2  // Converting a string to uppercase using a
3  // non-constant pointer to non-constant data.
4  #include <stdio.h>
5  #include <ctype.h>
6
7  void convertToUppercase(char *sPtr); // prototype
8
9  int main(void)
10 {
11    char string[] = "cHaRaCters and $32.98"; // initialize char array
12
13    printf("The string before conversion is: %s", string);
14    convertToUppercase(string);
15    printf("\nThe string after conversion is: %s\n", string);
16 }
17
```

Figure 7.10 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data (2 of 2)

```
18  // convert string to uppercase letters
19  void convertToUppercase(char *sPtr)
20  {
21     while (*sPtr != '\0') { // current character is not '\0'
22        *sPtr = toupper(*sPtr); // convert to uppercase
23        ++sPtr; // make sPtr point to the next character
24     }
25  }
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data (1 of 6)

- 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.

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data (2 of 6)

- For example, function `printCharacters` (Figure 7.11) declares parameter `sPtr` to be of type `const char *`
- The declaration is read from **right to left** as "`sPtr` is a pointer to a character constant." The function uses a `for` statement to output each character in the string until the null character is encountered.
- After each character is printed, pointer `sPtr` is incremented to point to the next character in the string.

Figure 7.11 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data (1 of 2)

```
1  // Fig. 7.11: fig07_11.c
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4
5  #include <stdio.h>
6
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    puts("The string is:");
15    printCharacters(string);
16    puts("");
17 }
18
```

Figure 7.11 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data (2 of 2)

```
19  // sPtr cannot be used to modify the character to which it points,
20  // i.e., sPtr is a "read-only" pointer
21  void printCharacters(const char *sPtr)
22  {
23     // loop through entire string
24     for (; *sPtr != '\0'; ++sPtr) { // no initialization
25        printf("%c", *sPtr);
26     }
27  }
```

```
The string is:
print characters of a string
```

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data (3 of 6)

- Figure 7.12 illustrates the attempt to compile a function that receives a non-constant pointer (`xPtr`) to constant data.
- This function attempts to modify the data pointed to by `xPtr`—which results in a compilation error.

Figure 7.12 Attempting to Modify Data Through a Non-Constant Pointer to Constant Data

```
1  // Fig. 7.12: fig07_12.c
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4  #include <stdio.h>
5  void f(const int *xPtr); // prototype
6
7  int main(void)
8  {
9     int y; // define y
10
11    f(&y); // f attempts illegal modification
12 }
13
14 // xPtr cannot be used to modify the
15 // value of the variable to which it points
16 void f(const int *xPtr)
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 }
```

```
error C2166: l-value specifies const object
```

7.6 Bubble Sort Using Pass-By-Reference (1 of 11)

- Let's improve the bubble sort program of Figure 6.15 to use two functions—`bubbleSort` and `swap`.
- Function **`bubbleSort`** sorts the array.
- It calls function `swap` to exchange the array elements `array[j]` and `array[j + 1]`
- Remember that C enforces **information hiding** between functions, so `swap` does not have access to individual array elements in `bubbleSort`.
- Because `bubbleSort` **wants** `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements **by reference** to `swap`—the **address** of each array element is passed explicitly.

7.6 Bubble Sort Using Pass-By-Reference (2 of 11)

- Although entire arrays are automatically passed by reference, individual array elements are scalars and are ordinarily passed by value.
- Therefore, `bubbleSort` uses the address operator (`&`) on each of the array elements in the `swap` call to effect pass-by-reference as follows
  - `swap(&array[j], &array[j + 1]);`
- Function `swap` receives `&array[j]` in pointer variable `element1Ptr`.

7.6 Bubble Sort Using Pass-By-Reference (3 of 11)

- Even though `swap`—because of information hiding—is **not** allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a **synonym** for `array[j]`—when `swap` references `*element1Ptr`, it's **actually** referencing `array[j]` in `bubbleSort`.
- Similarly, when `swap` references `*element2Ptr`, it's actually referencing `array[j+1]` in `bubbleSort`.

7.6 Bubble Sort Using Pass-By-Reference (4 of 11)

- Even though `swap` is not allowed to say

```
int hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

precisely the **same** effect is achieved by

```
int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
```

Figure 7.15 Putting Values into an Array, sorting the Values into Ascending Order and Printing the Resulting Array (1 of 4)

```
1  // Fig. 7.15: fig07_15.c
2  // Putting values into an array, sorting the values into
3  // ascending order and printing the resulting array.
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort(int * const array, const size_t 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    puts("Data items in original order");
15
16    // loop through array a
17    for (size_t i = 0; i < SIZE; ++i) {
18       printf("%4d", a[i]);
19    }
20
21    bubbleSort(a, SIZE); // sort the array
22
```

Figure 7.15 Putting Values into an Array, sorting the Values into Ascending Order and Printing the Resulting Array (2 of 4)

```
23      puts("\nData items in ascending order");
24
25      // loop through array a
26      for (size_t i = 0; i < SIZE; ++i) {
27         printf("%4d", a[i]);
28      }
29
30      puts("");
31   }
32
```

Figure 7.15 Putting Values into an Array, sorting the Values into Ascending Order and Printing the Resulting Array (3 of 4)

```
33  // sort an array of integers using bubble sort algorithm
34  void bubbleSort(int * const array, const size_t size)
35  {
36     void swap(int *element1Ptr, int *element2Ptr); // prototype
37
38     // loop to control passes
39     for (unsigned int pass = 0; pass < size - 1; ++pass) {
40
41        // loop to control comparisons during each pass
42        for (size_t j = 0; j < size - 1; ++j) {
43
44           // swap adjacent elements if they're out of order
45           if (array[j] > array[j + 1]) {
46              swap(&array[j], &array[j + 1]);
47           }
48        }
49     }
50  }
51
```

Figure 7.15 Putting Values into an Array, sorting the Values into Ascending Order and Printing the Resulting Array (4 of 4)

```
52  // swap values at memory locations to which element1Ptr and
53  // element2Ptr point
54  void swap(int *element1Ptr, int *element2Ptr)
55  {
56     int hold = *element1Ptr;
57     *element1Ptr = *element2Ptr;
58     *element2Ptr = hold;
59  }
```

```
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
```

7.6 Bubble Sort Using Pass-By-Reference (5 of 11)

- Several features of function `bubbleSort` should be noted.
- The function header declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a one-dimensional array as an argument (again, these notations are interchangeable).
- Parameter `size` is declared `const` to enforce the principle of least privilege.
- Although parameter `size` receives a copy of a value in `main`, and modifying the copy cannot change the value in `main`, `bubbleSort` does **not** need to alter `size` to accomplish its task.

7.6 Bubble Sort Using Pass-By-Reference (6 of 11)

- The size of the array remains fixed during the execution of function `bubbleSort`.
- Therefore, `size` is declared `const` to ensure that it's **not** modified.
- The prototype for function `swap` is included in the body of function `bubbleSort` because `bubbleSort` is the only function that calls `swap`.

7.6 Bubble Sort Using Pass-By-Reference (7 of 11)

- Placing the prototype in `bubbleSort` restricts proper calls of `swap` to those made from `bubbleSort`.
- Other functions that attempt to call `swap` do **not** have access to a proper function prototype, so the compiler generates one automatically.
- This normally results in a prototype that does **not** match the function header (and generates a compilation warning or error) because the compiler assumes `int` for the return type and the parameter types.

7.6 Bubble Sort Using Pass-By-Reference (8 of 11)

- Function `bubbleSort` receives the size of the array as a parameter
- The function must know the size of the array to sort the array.
- When an array is passed to a function, the memory address of the first element of the array is received by the function.
- The address, of course, does **not** convey the number of elements in the array.
- Therefore, you must pass the array size to the function.
- Another common practice is to pass a pointer to the beginning of the array and a pointer to the location just beyond the end of the array, the difference of the two pointers is the length of the array and the resulting code is simpler

7.6 Bubble Sort Using Pass-By-Reference (9 of 11)

- In the program, the size of the array is explicitly passed to function `bubbleSort`.
- There are two main benefits to this **approach—software reusability and proper software engineering**.
- By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts one-dimensional integer arrays of any size.

7.6 Bubble Sort Using Pass-By-Reference (10 of 11)

- We could have stored the array's size in a global variable that's accessible to the entire program.
- This would be more efficient, because a copy of the size is not made to pass to the function.
- However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs.

Software Engineering Observation 7.4

Global variables usually violate the principle of least privilege and can lead to poor software engineering. Global variables should be used only to represent truly shared resources, such as the time of day.

7.6 Bubble Sort Using Pass-By-Reference (11 of 11)

- The size of the array could have been programmed directly into the function.
- This restricts the use of the function to an array of a specific size and significantly reduces its reusability.
- Only programs processing one-dimensional integer arrays of the specific size coded into the function can use the function.

7.7 `sizeof` Operator (1 of 6)

- C provides the special unary operator **`sizeof`** to determine the size in bytes of an array (or any other data type).
- When applied to the name of an array as in Figure 7.16, the `sizeof` operator returns the total number of bytes in the array as type `size_t`.
- Variables of type `float` on this computer are stored in 4 bytes of memory, and `array` is defined to have 20 elements.
- Therefore, there are a total of 80 bytes in `array`.

Performance Tip 7.2

`sizeof` is a compile-time operator, so it does not incur any execution-time overhead.

Figure 7.16 Applying `Sizeof` to an Array Name Returns the Number of Bytes in the Array (1 of 2)

```
1  // Fig. 7.16: fig07_16.c
2  // Applying sizeof to an array name returns
3  // the number of bytes in the array.
4  #include <stdio.h>
5  #define SIZE 20
6
7  size_t getSize(float *ptr); // prototype
8
9  int main(void)
10 {
11    float array[SIZE]; // create array
12
13    printf("The number of bytes in the array is %u"
14           "\nThe number of bytes returned by getSize is %u\n",
15           sizeof(array), getSize(array));
16 }
17
18 // return size of ptr
19 size_t getSize(float *ptr)
20 {
21    return sizeof(ptr);
22 }
```

Figure 7.16 Applying `sizeof` to an Array Name Returns the Number of Bytes in the Array (2 of 2)

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

7.7 `sizeof` Operator (2 of 6)

- The number of elements in an array also can be determined with `sizeof`.
- For example, consider the following array definition:
  - `double real[22];`
- Variables of type `double` normally are stored in 8 bytes of memory.
- Thus, array `real` contains a total of 176 bytes.
- To determine the number of elements in the array, the following expression can be used:
  - `sizeof(real) / sizeof(real[0])`

7.7 `sizeof` Operator (3 of 6)

- The expression determines the number of bytes in array `real` and divides that value by the number of bytes used in memory to store the first element of array `real` (a `double` value).

7.7 `sizeof` Operator (4 of 6)

- Even though function `getSize` receives an array of 20 elements as an argument, the function's parameter `ptr` is simply a pointer to the array's first element.
- When you use `sizeof` with a pointer, it returns the **size of the pointer**, not the size of the item to which it points.
- The size of a pointer on our system is 4 bytes, so `getSize` returned 4.
- Also, the calculation shown above for determining the number of array elements using `sizeof` works only when using the actual array, not when using a pointer to the array.

7.7 `sizeof` Operator (5 of 6)

Determining the Sizes of the Standard Types, an Array and a Pointer

- Figure 7.17 calculates the number of bytes used to store each of the standard data types.
- The results of this program are implementation dependent and often differ across platforms and sometimes across different compilers on the same platform.

Figure 7.17 Using Operator `sizeof` to Determine Standard Data Type Sizes (1 of 2)

```
1  // Fig. 7.17: fig07_17.c
2  // Using operator sizeof to determine standard data type sizes.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    long long ll;
12    float f;
13    double d;
14    long double ld;
15    int array[20]; // create array of 20 int elements
16    int *ptr = array; // create pointer to array
17
18    printf("    sizeof c = %u\tsizeof(char)  = %u"
19           "\n     sizeof s = %u\tsizeof(short) = %u"
20           "\n     sizeof i = %u\tsizeof(int) = %u"
21           "\n     sizeof l = %u\tsizeof(long) = %u"
22           "\n    sizeof ll = %u\tsizeof(long long) = %u"
23           "\n     sizeof f = %u\tsizeof(float) = %u"
```

Figure 7.17 Using Operator `sizeof` to Determine Standard Data Type Sizes (2 of 2)

```
24              "\n     sizeof d = %u\tsizeof(double) = %u"
25              "\n    sizeof ld = %u\tsizeof(long double) = %u"
26              "\n sizeof array = %u"
27              "\n   sizeof ptr = %u\n",
28              sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
29              sizeof(int), sizeof l, sizeof(long), sizeof ll,
30              sizeof(long long), sizeof f, sizeof(float), sizeof d,
31              sizeof(double), sizeof ld, sizeof(long double),
32              sizeof array, sizeof ptr);
33  }
```

```
     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 ll = 8       sizeof(long long) = 8
     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
```

Portability Tip 7.1

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.

7.7 `sizeof` Operator (6 of 6)

- 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's 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 are required when a type is supplied as `size_of`'s operand.

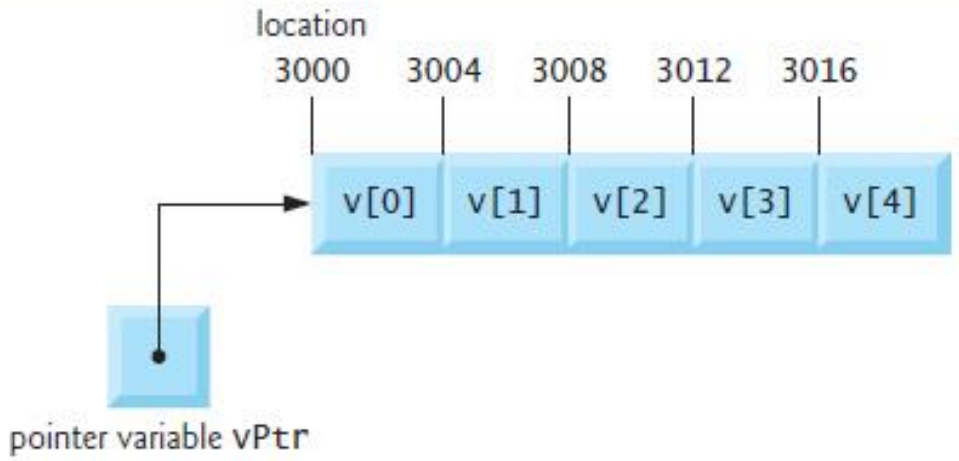
7.8 Pointer Expressions and Pointer Arithmetic (1 of 10)

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- However, not all the operators normally used in these expressions are valid in conjunction with pointer variables.
- This section describes the operators that can have pointers as operands, and how these operators are used.
- A limited set of arithmetic operations may be performed on pointers.
- A pointer may be **incremented** (++) or **decremented** (––) an integer may be **added** to a pointer (+ or +=), an integer may be **subtracted** from a pointer (– or –=) and one pointer may be subtracted from another—this last operation is meaningful only when **both** pointers point to elements of the **same** array.

7.8 Pointer Expressions and Pointer Arithmetic (2 of 10)

- 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`.
- Figure 7.18 illustrates this situation for a machine with 4-byte integers.
- Variable `vPtr` can be initialized to point to array `v` with either of the statements

Figure 7.18 Array v and a Pointer Variable vPtr that Points to v



7.8 Pointer Expressions and Pointer Arithmetic (3 of 10)

- In conventional arithmetic, `3000 + 2` yields the value `3002`.
- This is normally not the case with pointer arithmetic.
- When an integer is added to or subtracted from a pointer, the pointer is **not** incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.
- The number of bytes depends on the object's data type.
- For example, the statement
  - `vPtr += 2;`

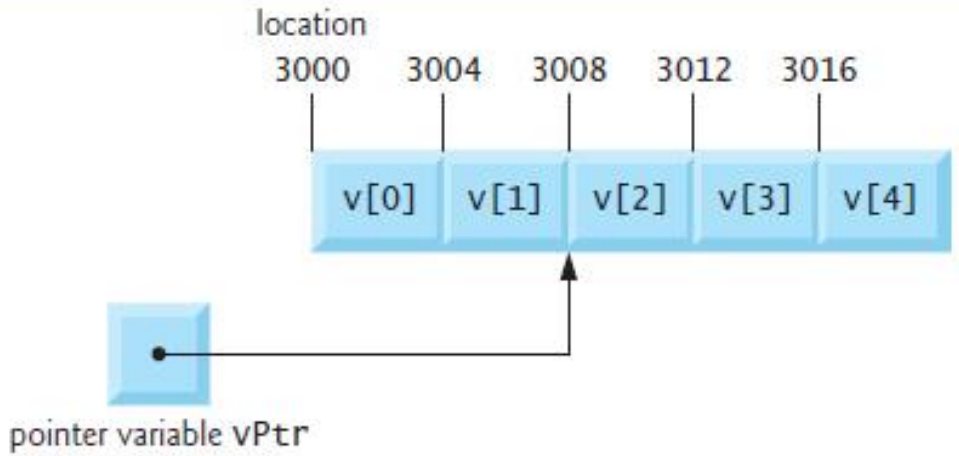
  would produce `3008` `(3000 + 2 * 4)`, assuming an integer is stored in 4 bytes of memory.

7.8 Pointer Expressions and Pointer Arithmetic (4 of 10)

- In the array `v`, `vPtr` would now point to `v[2]` (Figure 7.19).
- If an integer is stored in 2 bytes of memory, then the preceding calculation would result in memory location

  `3004 (3000 + 2 * 2)`.
- If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type.
- When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

Figure 7.19 The Pointer vPtr After Pointer Arithmetic



7.8 Pointer Expressions and Pointer Arithmetic (5 of 10)

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

  would set `vPtr` back to `3000`—the beginning of the array.
- If a pointer is being incremented or decremented by one, the increment (++) and decrement (––) operators can be used.
- Either of the statements
  - ```
    ++vPtr;
    vPtr++;
    ```

  increments the pointer to point to the **next** location in the array.
    ```

7.8 Pointer Expressions and Pointer Arithmetic (6 of 10)

- Either of the statements
  - `--vPtr;`
    `vPtr--;`

  decrements the pointer to point to the **previous** element of the array.
- Pointer variables may be subtracted from one another.

7.8 Pointer Expressions and Pointer Arithmetic (7 of 10)

- For example, if `vPtr` contains the location `3000`, and `v2Ptr` contains the address `3008`, the statement
  - `x = v2Ptr - vPtr;`

  would assign to `x` the **number of array elements** from `vPtr` to `v2Ptr`, in this case 2 (not 8).
- Pointer arithmetic is undefined unless performed on an array.
- We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.

Common Programming Error 7.5

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

Common Programming Error 7.6

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

7.8 Pointer Expressions and Pointer Arithmetic (8 of 10)

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

7.8 Pointer Expressions and Pointer Arithmetic (9 of 10)

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

Common Programming Error 7.7

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

Common Programming Error 7.9

Comparing two pointers that do not refer to elements in the **same** array.

7.8 Pointer Expressions and Pointer Arithmetic (10 of 10)

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

7.9 Relationship between Pointers and Arrays (1 of 11)

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An **array name** can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array indexing.
- Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined.
- Because the array name (without an index) 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;`

7.9 Relationship between Pointers and Arrays (2 of 11)

- This statement is equivalent to taking the address of the array's first element as follows:
  - `bPtr = &b[0];`
- Array element `b[3]` can alternatively be referenced with the pointer expression
  - `*(bPtr + 3)`
- The `3` in the expression is the **offset** to the pointer.
- When the pointer points to the array's first element, the offset indicates which array element should be referenced, and the offset value is identical to the array index.
- This notation is referred to as **pointer/offset notation**.

7.9 Relationship between Pointers and Arrays (3 of 11)

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

  can be written with the pointer expression
  - `bPtr + 3`
- The array itself can be treated as a pointer and used in pointer arithmetic.

7.9 Relationship between Pointers and Arrays (4 of 11)

- For example, the expression
  - `*(b + 3)`

  also refers to the array element `b[3]`.
- In general, all indexed array expressions can be written with a pointer and an offset.
- In this case, pointer/offset notation was used with the name of the array as a pointer.
- The preceding statement does not modify the array name in any way; `b` still points to the first element in the array.
- Pointers can be indexed like arrays.

7.9 Relationship between Pointers and Arrays (5 of 11)

- If `bPtr` has the value `b`, the expression
  - `bPtr[1]`

  refers to the array element `b[1]`.
- This is referred to as **pointer/index notation**.
- Remember that an array name is essentially a constant pointer; it always points to the beginning of the array.
- Thus, the expression
  - `b += 3`

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

7.9 Relationship between Pointers and Arrays (6 of 11)

- Figure 7.20 uses the four methods we've discussed for referring to array elements—array indexing, pointer/offset with the array name as a pointer, **pointer indexing**, and pointer/offset with a pointer—to print the four elements of the integer array b.

Figure 7.20 Using Indexing and Pointer Notations with Arrays (1 of 3)

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

Figure 7.20 Using Indexing and Pointer Notations with Arrays (2 of 3)

```
23      // loop through array b
24      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
25         printf("*(b + %u) = %d\n", offset, *(b + offset));
26      }
27
28      // output array b using bPtr and array index notation
29      puts("\nPointer index notation");
30
31      // loop through array b
32      for (size_t i = 0; i < ARRAY_SIZE; ++i) {
33         printf("bPtr[%u] = %d\n", i, bPtr[i]);
34      }
35
36      // output array b using bPtr and pointer/offset notation
37      puts("\nPointer/offset notation");
38
39      // loop through array b
40      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
41         printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
42      }
43   }
```

Figure 7.20 Using Indexing and Pointer Notations with Arrays (3 of 3)

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

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

7.9 Relationship between Pointers and Arrays (7 of 11)

String Copying with Arrays and Pointers

- 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 Figure 7.21.
- Both functions copy a string into a character array.
- After a comparison of the function prototypes for `copy1` and `copy2`, the functions appear identical.
- They accomplish the same task; but they're implemented differently.

Figure 7.21 Copying a String Using Array Notation and Pointer Notation (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  #define SIZE 10
5
6  void copy1(char * const s1, const char * const s2); // prototype
7  void copy2(char *s1, const char *s2); // prototype
8
9  int main(void)
10 {
11    char string1[SIZE]; // create array string1
12    char *string2 = "Hello"; // create a pointer to a string
13
14    copy1(string1, string2);
15    printf("string1 = %s\n", string1);
16
17    char string3[SIZE]; // create array string3
18    char string4[] = "Good Bye"; // create an array containing a string
19
20    copy2(string3, string4);
21    printf("string3 = %s\n", string3);
22 }
23
```

Figure 7.21 Copying a String Using Array Notation and Pointer Notation (2 of 2)

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

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

7.9 Relationship between Pointers and Arrays (8 of 11)

- Function `copy1` uses **array index notation** to copy the string in `s2` to the character array `s1`.
- The function defines counter variable `i` as the array index.
- The `for` statement header performs the entire copy operation—its body is the empty statement.
- The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop.
- The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`.
- When the null character is encountered in `s2`, it's assigned to `s1`, and the value of the assignment becomes the value assigned to the left operand (`s1`).

7.9 Relationship between Pointers and Arrays (9 of 11)

- The loop terminates when the null character is assigned from `s1` to `s2` (false).
- Function `copy2` uses **pointers and pointer arithmetic** to copy the string in `s2` to the character array `s1`.
- Again, the `for` statement header performs the entire copy operation.
- The header does not include any variable initialization.
- As in function `copy1`, the expression `(*s1 = *s2)` performs the copy operation.
- Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`.

7.9 Relationship between Pointers and Arrays (10 of 11)

- After the assignment in the condition, the pointers are incremented to point to the next element of array `s1` and the next character of string `s2`, respectively.
- When the null character is encountered in `s2`, it's assigned to the dereferenced pointer `s1` and the loop terminates.
- **The first argument to both `copy1` and `copy2` must be an array large enough to hold the string in the second argument.**
- Otherwise, an error may occur when an attempt is made to write into a memory location that's not part of the array.
- Also, the second parameter of each function is declared as `const` char * (a constant string).

7.9 Relationship between Pointers and Arrays (11 of 11)

- In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are **never modified**.
- Therefore, the second parameter is declared to point to a constant value so that the **principle of least privilege** is enforced—neither function requires the capability of modifying the second argument, so neither function is provided with that capability.

7.10 Arrays of Pointers (1 of 4)

- Arrays may contain pointers.
- A common use of an **array of pointers** is to form an **array of strings**, referred to simply as a **string array**.
- Each entry in the array is a string, but in C a string is essentially a pointer to its first character.
- So each entry in an array of strings is actually a pointer to the first character of a string.
- Consider the definition of string array `suit`, which might be useful in representing a deck of cards.
  - `const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};`

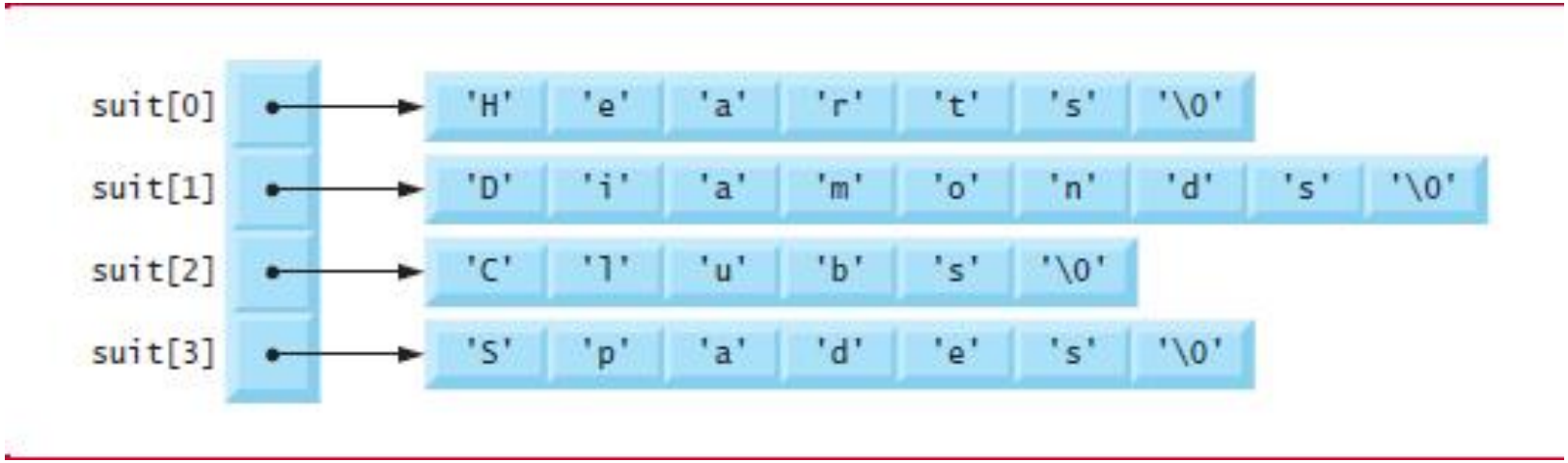
7.10 Arrays of Pointers (2 of 4)

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

7.10 Arrays of Pointers (3 of 4)

- The four strings are 7, 9, 6 and 7 characters long, respectively.
- Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array (Figure 7.22).
- Each pointer points to the first character of its corresponding string.
- Thus, even though the `suit` array is **fixed** in size, it provides access to character strings of **any length**.
- This flexibility is one example of C's powerful data-structuring capabilities.

Figure 7.22 Graphical Representation of the `Suit` Array



7.10 Arrays of Pointers (4 of 4)

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

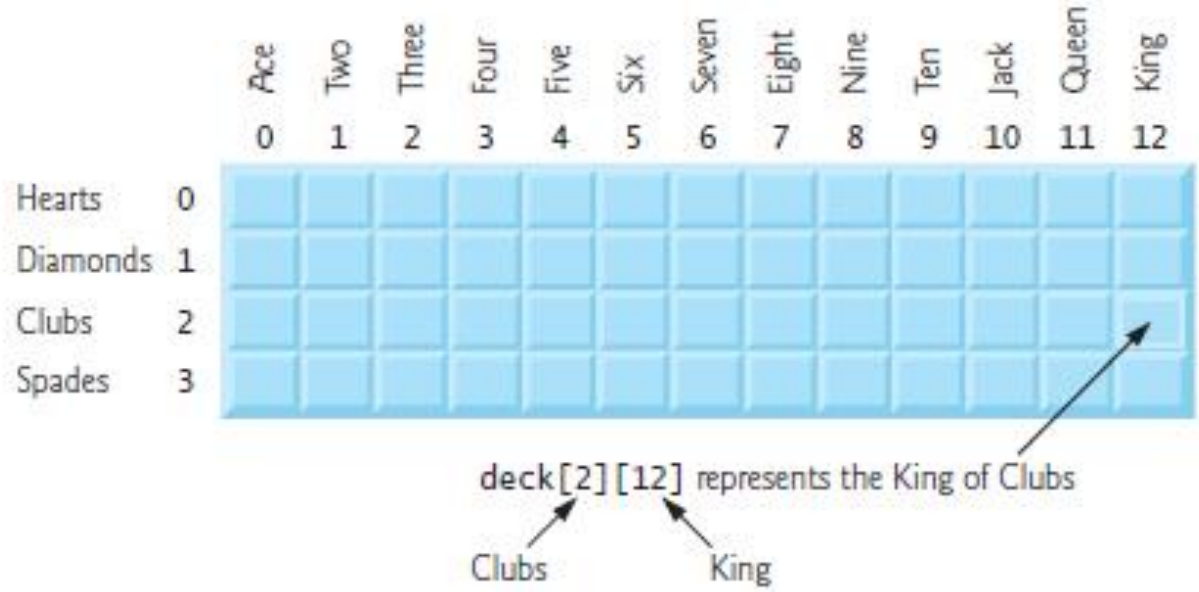
7.11 Case Study: Card Shuffling and Dealing Simulation (1 of 14)

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

7.11 Case Study: Card Shuffling and Dealing Simulation (2 of 14)

- The top-down approach is particularly useful in attacking larger, more complex problems than you've seen in earlier chapters.
- We use 4-by-13 two-dimensional array `deck` to represent the deck of playing cards (Figure 7.23).
- The rows correspond to the **suits**—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades.
- The columns correspond to the **face** values of the cards—0 through 9 correspond to ace through ten, and columns 10 through 12 correspond to jack, queen and king.
- We shall load string array `suit` with character strings representing the four suits, and string array `face` with character strings representing the thirteen face values.

Figure 7.23 Two-Dimensional Array Representation of a Deck of Cards



7.11 Case Study: Card Shuffling and Dealing Simulation (3 of 14)

- This simulated deck of cards may be **shuffled** as follows.
- First the array `deck` is cleared to zeros.
- Then, a `row(0-3)` and a `column(0-12)` are each chosen **at random**.
- The number 1 is inserted in array element `deck[row][column]` to indicate that this card will be the first one dealt from the shuffled deck.
- This process continues with the numbers 2, 3, …, 52 being randomly inserted in the `deck` array to indicate which cards are to be placed second, third, …, and fifty-second in the shuffled deck.

7.11 Case Study: Card Shuffling and Dealing Simulation (4 of 14)

- As the `deck` array begins to fill with card numbers, it's possible that a card will be selected again—i.e., `deck[row][column]` will be nonzero when it's selected.
- This selection is simply ignored and other `rows` and `columns` are repeatedly chosen at random until an **unselected** card is found.
- Eventually, the numbers 1 through 52 will occupy the 52 slots of the `deck` array.
- At this point, the deck of cards is fully shuffled.

7.11 Case Study: Card Shuffling and Dealing Simulation (5 of 14)

- This shuffling algorithm can execute **indefinitely** if cards that have already been shuffled are repeatedly selected at random.
- This phenomenon is known as **indefinite postponement**.
- In this chapter's exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

7.11 Case Study: Card Shuffling and Dealing Simulation (6 of 14)

- To deal the first card, we search the array for `deck[row][column]` equal to `1`.
- This is accomplished with nested `for` statements that vary `row` from 0 to 3 and `column` from 0 to 12.
- What card does that element of the array correspond to?
- The `suit` array has been preloaded with the four suits, so to get the suit, we print the character string `suit[row].`
- Similarly, to get the face value of the card, we print the character string `face[column].`
- We also print the character string "`of`".

7.11 Case Study: Card Shuffling and Dealing Simulation (7 of 14)

- Printing this information in the proper order enables us to print each card in the form "`King of Clubs`", "`Ace of Diamonds`" and so on.
- Let's proceed with the top-down, stepwise refinement process.
- The **top** is simply
  - Shuffle and deal 52 cards
- Our **first** refinement yields:
  - Initialize the suit array

    Initialize the face array

    Initialize the deck array

    Shuffle the deck

    Deal 52 cards

7.11 Case Study: Card Shuffling and Dealing Simulation (8 of 14)

- "Shuffle the deck" may be expanded as follows:
  - For each of the 52 cards

    Place card number in randomly selected unoccupied slot of deck
- "Deal 52 cards" may be expanded as follows:
  - For each of the 52 cards

    Find card number in deck array and print face and suit of card

7.11 Case Study: Card Shuffling and Dealing Simulation (9 of 14)

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

    Initialize the face array
    Initialize the deck array

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

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

7.11 Case Study: Card Shuffling and Dealing Simulation (10 of 14)

- "Place card number in randomly selected unoccupied slot of deck" may be expanded as:
  - Choose slot of deck randomly

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

    Place card number in chosen slot of deck

- "Find card number in deck array and print face and suit of card" may be expanded as:
  - For each slot of the deck array
    If slot contains card number
    Print the face and suit of the card

7.11 Case Study: Card Shuffling and Dealing Simulation (11 of 14)

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

    For each of the 52 cards
    Choose slot of deck randomly

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

    Place card number in chosen slot of deck

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

7.11 Case Study: Card Shuffling and Dealing Simulation (12 of 14)

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

7.11 Case Study: Card Shuffling and Dealing Simulation (13 of 14)

- The card shuffling and dealing program is shown in Figure 7.24, and a sample execution is shown in Figure 7.25.
- Conversion specifier `%s` is used to print strings of characters in the calls to `printf`.
- The corresponding argument in the `printf` call must be a pointer to `char` (or a `char` array).
- The format specification `"%5s of %-8s"` prints a character string **right justified** in a field of five characters followed by `"of"` and a character string **left justified** in a field of eight characters.
- The **minus sign** in `%-8s` signifies left justification.

Figure 7.24 Card Shuffling and Dealing (1 of 4)

```
1  // Fig. 7.24: fig07_24.c
2  // Card shuffling and dealing.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define SUITS 4
8  #define FACES 13
9  #define CARDS 52
10
11 // prototypes
12 void shuffle(unsigned int wDeck[][FACES]); // shuffling modifies wDeck
13 void deal(unsigned int wDeck[][FACES], const char *wFace[],
14    const char *wSuit[]); // dealing doesn't modify the arrays
15
16 int main(void)
17 {
18    // initialize deck array
19    unsigned int deck[SUITS][FACES] = {0};
20
21    srand(time(NULL)); // seed random-number generator
22    shuffle(deck); // shuffle the deck
23
```

Figure 7.24 Card Shuffling and Dealing (2 of 4)

```
24      // initialize suit array
25      const char *suit[SUITS] =
26         {"Hearts", "Diamonds", "Clubs", "Spades"};
27
28      // initialize face array
29      const char *face[FACES] =
30         {"Ace", "Deuce", "Three", "Four",
31          "Five", "Six", "Seven", "Eight",
32          "Nine", "Ten", "Jack", "Queen", "King"};
33
34      deal(deck, face, suit); // deal the deck
35   }
36
```

Figure 7.24 Card Shuffling and Dealing (3 of 4)

```
37  // shuffle cards in deck
38  void shuffle(unsigned int wDeck[][FACES])
39  {
40     // for each of the cards, choose slot of deck randomly
41     for (size_t card = 1; card <= CARDS; ++card) {
42        size_t row; // row number
43        size_t column; // column number
44
45        // choose new random location until unoccupied slot found
46        do {
47           row = rand() % SUITS;
48           column = rand() % FACES;
49        } while(wDeck[row][column] != 0);
50
51        // place card number in chosen slot of deck
52        wDeck[row][column] = card;
53     }
54  }
55
```

Figure 7.24 Card Shuffling and Dealing (4 of 4)

```
56  // deal cards in deck
57  void deal(unsigned int wDeck[][FACES], const char *wFace[],
58     const char *wSuit[])
59  {
60     // deal each of the cards
61     for (size_t card = 1; card <= CARDS; ++card) {
62        // loop through rows of wDeck
63        for (size_t row = 0; row < SUITS; ++row) {
64           // loop through columns of wDeck for current row
65           for (size_t column = 0; column < FACES; ++column) {
66              // if slot contains current card, display card
67              if (wDeck[row][column] == card) {
68                 printf("%5s of %-8s%c", wFace[column], wSuit[row],
69                    card % 2 == 0 ? '\n' : '\t'); // 2-column format
70              }
71           }
72        }
73     }
74  }
```

Figure 7.25 Sample Run of Card Dealing Program

```
 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
```

7.11 Case Study: Card Shuffling and Dealing Simulation (14 of 14)

- There's a weakness in the dealing algorithm.
- Once a match is found, the two inner `for` statements continue searching the remaining elements of `deck` for a match.
- We correct this deficiency in this chapter's exercises and in a Chapter 10 case study.

Copyright