8

Pointers and Pointer-Based Strings

Addresses are given to us to conceal our whereabouts.

— Saki (H. H. Munro)

By indirection find direction out.

— William Shakespeare

Many things, having full reference
To one consent, may work contrariously.

— William Shakespeare

You will find it a very good practice always to verify your references, sir!

— Dr. Routh

OBJECTIVES

In this chapter you will learn:

- What pointers are.
- The similarities and differences between pointers and references and when to use each.
- To use pointers to pass arguments to functions by reference.
- To use pointer-based C-style strings.
- The close relationships among pointers, arrays and C-style strings.
- To use pointers to functions.
- To declare and use arrays of C-style strings.

Outline

8.1 Introduction

- **Pointers**
  - **Powerful, but difficult to master**
  - **Can be used to perform pass-by-reference**
  - **Can be used to create and manipulate dynamic data structures**
  - **Close relationship with arrays and strings**
    - `char *` **pointer-based strings**

8.2 Pointer Variable Declarations and Initialization

- **Pointer variables**
  - **Contain memory addresses as values**
    - **Normally, variable contains specific value (direct reference)**
    - **Pointers contain address of variable that has specific value (indirect reference)**
- **Indirection**
  - **Referencing value through pointer**

8.2 Pointer Variable Declarations and Initialization (Cont.)

- **Pointer declarations**
  - *** indicates variable is a pointer**
    - **Example**
      - `int *myPtr;`
        - **Declares pointer to `int`, of type `int *`**
    - **Multiple pointers require multiple asterisks**

      ```
      int *myPtr1, *myPtr2;
      ```
- **Pointer initialization**
  - **Initialized to `0`, `NULL`, or an address**
    - **`0` or `NULL` points to nothing (null pointer)**

Common Programming Error 8.1

Assuming that the * used to declare a pointer distributes to all variable names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the * prefixed to the name (either with or without a space in between—the compiler ignores the space). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.

Good Programming Practice 8.1

Although it is not a requirement, including the letters `Ptr` in pointer variable names makes it clear that these variables are pointers and that they must be handled appropriately.

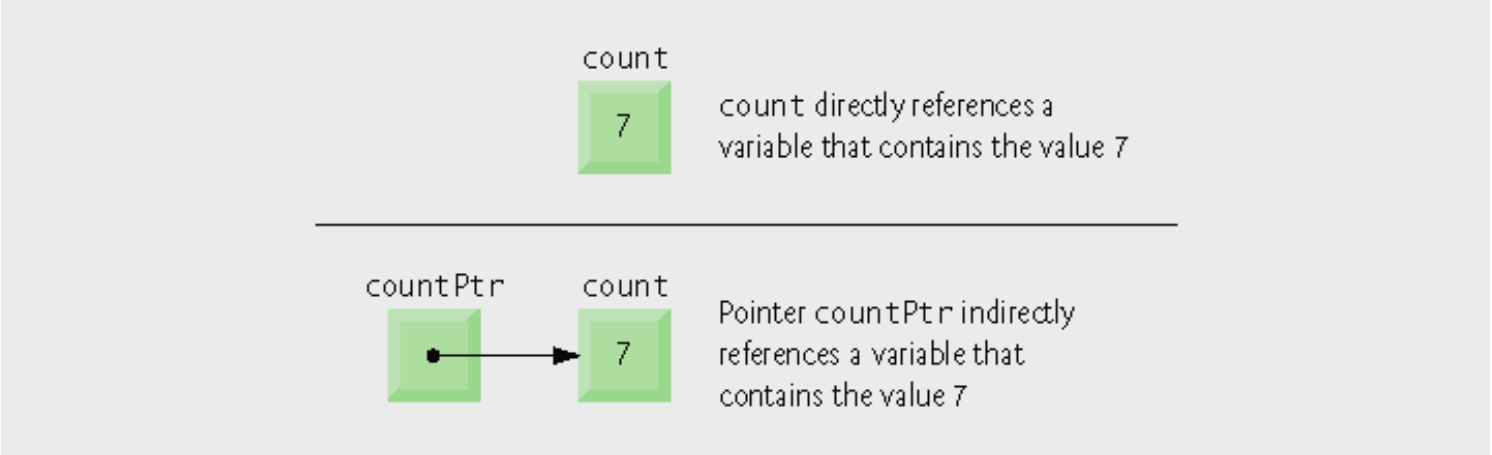


Fig. 8.1 | Directly and indirectly referencing a variable.

Error-Prevention Tip 8.1

Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

8.3 Pointer Operators

- **Address operator (&)**
  - **Returns memory address of its operand**
  - **Example**
    - `int y = 5;`
      `int *yPtr;`
      `yPtr = &y;`
      assigns the address of variable `y` to pointer variable `yPtr`
      - **Variable `yPtr` "points to" `y`**
        - **`yPtr` indirectly references variable `y`'s value**

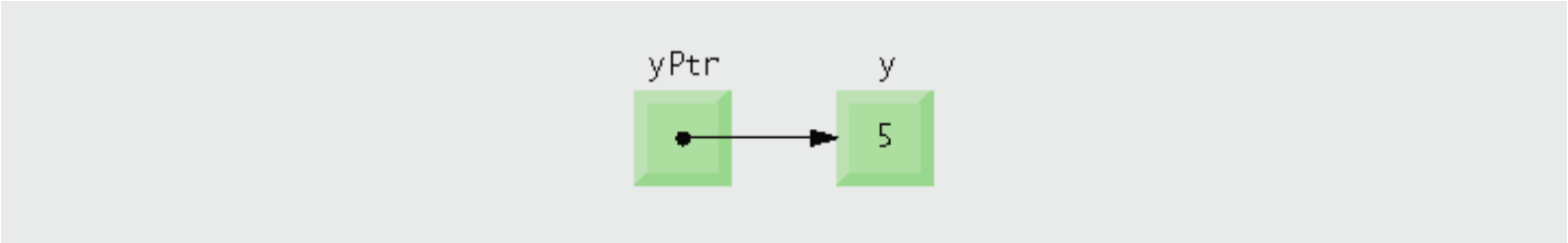


Fig. 8.2 | Graphical representation of a pointer pointing to a variable in memory.

8.3 Pointer Operators (Cont.)

- * operator
  - Also called indirection operator or dereferencing operator
  - Returns synonym for the object its operand points to
  - `*yPtr` returns `y` (because `yPtr` points to `y`)
  - Dereferenced pointer is an *lvalue*

```
*yptr = 9;
```

- * and & are inverses of each other
  - Will "cancel one another out" when applied consecutively in either order

Fig. 8.3 | Representation of `y` and `yPtr` in memory.

Common Programming Error 8.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.

Common Programming Error 8.4

Dereferencing a null pointer is normally a fatal execution-time error.

Portability Tip 8.1

The format in which a pointer is output is compiler dependent. Some systems output pointer values as hexadecimal integers, while others use decimal integers.

Outline

fig08_04.cpp

(1 of 2)

```
1  // Fig. 8.4: fig08_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
```

Variable **aPtr** is a point to an **int**

Initialize **aPtr** with the address of variable **a**

```
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

Address of **a** and the value of **aPtr** are identical

Value of **a** and the dereferenced **aPtr** are identical

***** and **&** are inverses of each other

```
The address of a is 0012F580
The value of aPtr is 0012F580

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

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

***** and **&** are inverses; same result when both are applied to **aPtr**

Operators	Associativity	Type
`()` `[]`	left to right	highest
`++` `--` `static_cast<` *type* `>(` *operand* `)`	left to right	unary (postfix)
`++` `--` `+` `-` `!` `&` `*`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	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. 8.5 | Operator precedence and associativity.

8.4 Passing Arguments to Functions by Reference with Pointers

- Three ways to pass arguments to a function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- A function can `return` only one value
- Arguments passed to a function using reference arguments
  - Function can modify original values of arguments
    - More than one value "returned"

8.4 Passing Arguments to Functions by Reference with Pointers (Cont.)

- **Pass-by-reference with pointer arguments**
  - **Simulates pass-by-reference**
    - **Use pointers and indirection operator**
  - **Pass address of argument using & operator**
  - **Arrays not passed with & because array name is already a pointer**
  - *** operator used as alias/nickname for variable inside of function**

Outline

fig08_06.cpp

(1 of 1)

```
1  // Fig. 8.6: fig08_06.cpp
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << numb
14
15    number = cubeByValue( number ); // pass number by value to cubeByValue
16    cout << "\nThe new value of number is " << number << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer ar
21 int cubeByValue( int n )
22 {
23    return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue
```

Pass number by value; result returned by **cubeByValue**

cubeByValue receives parameter passed-by-value

Cubes local variable **n** and **return** the result

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

Common Programming Error 8.5

Not dereferencing a pointer when it is necessary to do so to obtain the value to which the pointer points is an error.

Outline

fig08_07.cpp

(1 of 1)

```
1  // Fig. 8.7: fig08_07.cpp
2  // Cube a variable using pass-by-reference with a pointer argument.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is "
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
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
```

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

Prototype indicates parameter is a pointer to an **int**

Apply address operator **&** to pass address of **number** to **cubeByReference**

cubeByReference modifies variable **number**

Modify and access **int** variable using indirection operator ***

cubeByReference receives address of an **int** variable, i.e., a pointer to an **int**

Software Engineering Observation 8.1

Use pass-by-value to pass arguments to a function unless the caller explicitly requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.

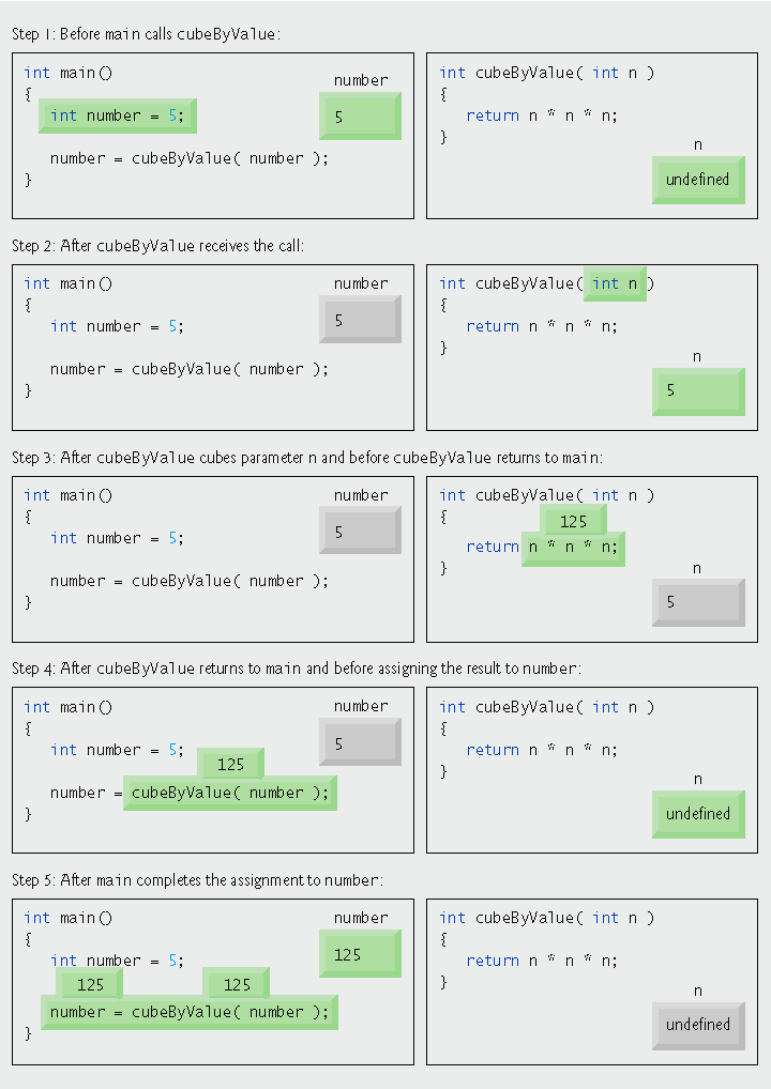


Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6.

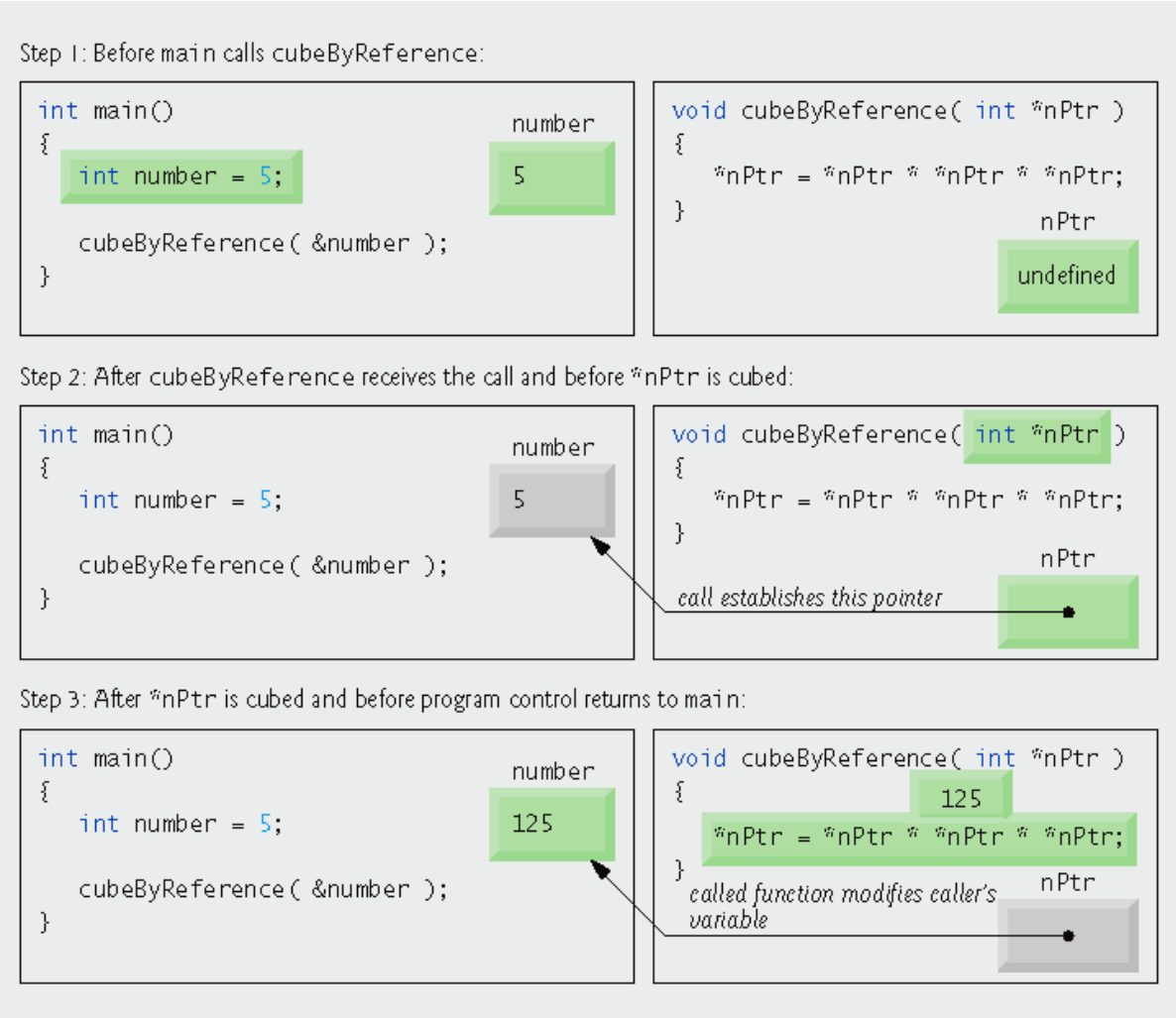


Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7.

8.5 Using const with Pointers

- const qualifier
  - Indicates that value of variable should not be modified
  - const used when function does not need to change the variable's value
- Principle of least privilege
  - Award function enough access to accomplish task, but no more
  - Example
    - A function that prints the elements of an array, takes array and int indicating length
      - Array contents are not changed – should be const
      - Array length is not changed – should be const

Portability Tip 8.2

Although `const` is well defined in ANSI C and C++, some compilers do not enforce it properly. So a good rule is, "Know your compiler."

Software Engineering Observation 8.2

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

Error-Prevention Tip 8.2

Before using a function, check its function prototype to determine the parameters that it can modify.

8.5 Using const with Pointers (Cont.)

- **Four ways to pass pointer to function**
  - **Nonconstant pointer to nonconstant data**
    - **Highest amount of access**
    - **Data can be modified through the dereferenced pointer**
    - **Pointer can be modified to point to other data**
      - **Pointer arithmetic**
        - **Operator ++ moves array pointer to the next element**
    - **Its declaration does not include `const` qualifier**

Outline

fig08_10.cpp

(1 of 2)

```
1  // Fig. 8.10: fig08_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // prototypes for islower and
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indicates successful termination
22 } // end main
```

Parameter is a nonconstant pointer to nonconstant data

convertToUppercase modifies variable **phrase**

```
23
24 // convert string to uppercase letters
25 void convertToUppercase( char *sPtr )
26 {
27    while ( *sPtr != '\0' ) // loop while current character is not '\0'
28    {
29       if ( islower( *sPtr ) ) // if character
30          *sPtr = toupper( *sPtr ); // convert
31
32       sPtr++; // move sPtr to next character in st
33    } // end while
34 } // end function convertToUppercase
```

```
The phrase before conversion is: characters and $32.
The phrase after conversion is:  CHARACTERS AND $32.98
```

Parameter **sPtr** is a nonconstant pointer to nonconstant data

Function **islower** returns **true** if the character is lowercase

Function **toupper** returns corresponding uppercase character if original character is lowercase; otherwise **toupper** returns the original character

Modify the memory address stored in **sPtr** to point to the next element of the array

8.5 Using const with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Nonconstant pointer to constant data**
    - **Pointer can be modified to point to any appropriate data item**
    - **Data cannot be modified through this pointer**
    - **Provides the performance of pass-by-reference and the protection of pass-by-value**

Outline

fig08_11.cpp

(1 of 1)

```
1  // Fig. 8.11: fig08_11.cpp
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // print using pointer to const data
9
10 int main()
11 {
12    const char phrase[] = "print characters of a
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // print characters in phrase
16    cout << endl;
17    return 0; // indicates successful terminati
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
25       cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

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

Parameter is a nonconstant pointer to constant data

Pass pointer **phrase** to function **printCharacters**

sPtr is a nonconstant pointer to constant data; it cannot modify the character to which it points

Increment **sPtr** to point to the next character

Outline

fig08_12.cpp

(1 of 2)

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12    return 0; // indicates successful termination
13 } // end main
```

Parameter is a nonconstant pointer to constant data

Pass the address of **int** variable **y** to attempt an illegal modification

Outline

```
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

Attempt to modify a **const** object pointed to by **xPtr**

Borland C++ command-line compiler error messa

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C2166: l-value specifies const object
```

Error produced when attempting to compile

GNU C++ compiler error message:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

fig08_12.cpp

(2 of 2)

Performance Tip 8.1

If they do not need to be modified by the called function, pass large objects using pointers to constant data or references to constant data, to obtain the performance benefits of pass-by-reference.

Software Engineering Observation 8.3

Pass large objects using pointers to constant data, or references to constant data, to obtain the security of pass-by-value.

8.5 Using const with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Constant pointer to nonconstant data**
    - **Always points to the same memory location**
      - **Can only access other elements using subscript notation**
    - **Data can be modified through the pointer**
    - **Default for an array name**
      - **Can be used by a function to receive an array argument**
    - **Must be initialized when declared**

Outline

fig08_13.cpp

(1 of 1)

```
1  // Fig. 8.13: fig08_13.cpp
2  // Attempting to modify a constant pointer to non-constant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must b
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15    return 0; // indicates successful termination
16 } // end main
```

ptr is a constant pointer to an integer

Can modify **x** (pointed to by **ptr**) since **x** is not constant

Cannot modify **ptr** to point to a new address since **ptr** is constant

Borland C++ command-line compiler error message:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()s
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Line 14 generates a compiler error by attempting to assign a new address to a constant pointer

Common Programming Error 8.6

Not initializing a pointer that is declared `const` is a compilation error.

8.5 Using const with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Constant pointer to constant data**
    - **Least amount of access**
    - **Always points to the same memory location**
    - **Data cannot be modified using this pointer**

Outline

fig08_14.cpp

(1 of 2)

```
1  // Fig. 8.14: fig08_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr is a constant pointer to a constant integer.
12    // ptr always points to the same location; the integer
13    // at that location cannot be modified.
14    const int *const ptr = &x;
15
16    cout << *ptr << endl;
17
18    *ptr = 7; // error: *ptr is const; cannot assign new value
19    ptr = &y; // error: ptr is const; cannot assign new address
20    return 0; // indicates successful termination
21 } // end main
```

ptr is a constant pointer to a constant integer

Cannot modify **x** (pointed to by **ptr**) since ***ptr** is constant

Cannot modify **ptr** to point to a new address since **ptr** is constant

Outline

Borland C++ command-line compiler error message:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
   l-value specifies const object
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

Line 18 generates a compiler

Line 19 generates a compiler error by attempting to assign a new address to a constant pointer

8.6 Selection Sort Using Pass-by-Reference

- Implement `selectionSort` using pointers
  - Selection sort algorithm
    - Swap smallest element with the first element
    - Swap second-smallest element with the second element
    - Etc.
  - Want function `swap` to access array elements
    - Individual array elements: scalars
      - Passed by value by default
    - Pass by reference via pointers using address operator &

Outline

fig08_15.cpp

(1 of 3)

```
1  // Fig. 8.15: fig08_15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order and prints the resulting array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // prototype
12 void swap( int * const, int * const ); // prototype
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // sort the array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
```

```
30
31    cout << endl;
32    return 0; // indicates successful termination
33 } // end main
34
35 // function to sort an array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // index of smallest element
39
40    // loop over size - 1 elements
41    for ( int i = 0; i < size - 1; i++ )
42    {
43       smallest = i; // first index of remaining array
44
45       // loop to find index of smallest element
46       for ( int index = i + 1; index < size; index++ )
47
48          if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51       swap( &array[ i ], &array[ smallest ] );
52    } // end if
53 } // end function selectionSort
```

Declare **array** as **int *array** (rather than **int array[]**) to indicate function **selectionSort** receives single-subscripted array

Receives the size of the array as an argument; declared **const** to ensure that **size** is not modified

Outline

fig08_15.cpp

```
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59    int hold = *element1Ptr;
60    *element1Ptr = *element2Ptr;
61    *element2Ptr = hold;
62 } // end function swap
```

```
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
```

Arguments are assed by reference, allowing the function to swap values at the original memory locations

Software Engineering Observation 8.4

When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size). This makes the function more reusable.

8.7 sizeof Operators

- sizeof operator
  - Returns size of operand in bytes
  - For arrays, sizeof returns

    (size of 1 element) * (number of elements)
  - If sizeof(int) returns 4 then

```
int myArray[ 10 ];
cout << sizeof( myArray );
```

    will print 40
  - Can be used with
    - Variable names
    - Type names
    - Constant values

Common Programming Error 8.7

Using the `sizeof` operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.

Outline

fig08_16.cpp

(1 of 1)

```
1  // Fig. 8.16: fig08_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  size_t getSize( double * ); // prototype
9
10 int main()
11 {
12    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
13
14    cout << "The number of bytes in the array is " << sizeof( array );
15
16    cout << "\nThe number of bytes returned by getSize is "
17       << getSize( array ) << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // return size of ptr
22 size_t getSize( double *ptr )
23 {
24    return sizeof( ptr );
25 } // end function getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Operator **sizeof** applied to an array returns total number of bytes in the array

Function **getSize** returns the number of bytes used to store **array** address

Operator **sizeof** returns number of bytes of pointer

8.7 sizeof Operators (Cont.)

- sizeof operator (Cont.)
  - Is performed at compiler-time
  - For `double realArray[ 22 ];`
    - Use `sizeof realArray / sizeof( double )` to calculate the number of elements in `realArray`
  - Parentheses are only required if the operand is a type name

Outline

fig08_17.cpp

(1 of 2)

```
1  // Fig. 8.17: fig08_17.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char c; // variable of type char
10    short s; // variable of type short
11    int i; // variable of type int
12    long l; // variable of type long
13    float f; // variable of type float
14    double d; // variable of type double
15    long double ld; // variable of type long double
16    int array[ 20 ]; // array of int
17    int *ptr = array; // variable of type int *
```

Outline

```
18
19    cout << "sizeof c = " << sizeof c
20       << "\tsizeof(char) = " << sizeof( char )
21       << "\nsizeof s = " << sizeof s
22       << "\tsizeof(short) = " << sizeof( short )
23       << "\nsizeof i = " << sizeof i
24       << "\tsizeof(int) = " << sizeof( int )
25       << "\nsizeof l = " << sizeof l
26       << "\tsizeof(long) = " << sizeof( long )
27       << "\nsizeof f = " << sizeof f
28       << "\tsizeof(float) = " << sizeof( float )
29       << "\nsizeof d = " << sizeof d
30       << "\tsizeof(double) = " << sizeof( double )
31       << "\nsizeof ld = " << sizeof ld
32       << "\tsizeof(long double) = " << sizeof( long double )
33       << "\nsizeof array = " << sizeof array
34       << "\nsizeof ptr = " << sizeof ptr << endl;
35    return 0; // indicates successful termination
36 } // end main
```

Operator **sizeof** can be used on a variable name

Operator **sizeof** can be used on a type name

7.cpp

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

Operator **sizeof** returns the total number of bytes in the array

Portability Tip 8.3

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.

Common Programming Error 8.8

Omitting the parentheses in a `sizeof` operation when the operand is a type name is a compilation error.

Performance Tip 8.2

Because `sizeof` is a compile-time unary operator, not an execution-time operator, using `sizeof` does not negatively impact execution performance.

Error-Prevention Tip 8.3

To avoid errors associated with omitting the parentheses around the operand of operator `sizeof`, many programmers include parentheses around every `sizeof` operand.

8.8 Pointer Expressions and Pointer Arithmetic

- **Pointer arithmetic**
  - **Increment/decrement pointer (++ or --)**
  - **Add/subtract an integer to/from a pointer (+ or +=, - or -=)**
  - **Pointers may be subtracted from each other**
  - **Pointer arithmetic is meaningless unless performed on a pointer to an array**

8.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- 5 element `int` array on a machine using 4 byte `int`s
  - `vPtr` points to first element `v[ 0 ]`, at location 3000

    `vPtr = &v[ 0 ];`
  - `vPtr += 2;` sets `vPtr` to 3008 (3000 + 2 * 4)

    `vPtr` points to `v[ 2 ]`
- Subtracting pointers
  - Returns number of elements between two addresses

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

Portability Tip 8.4

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

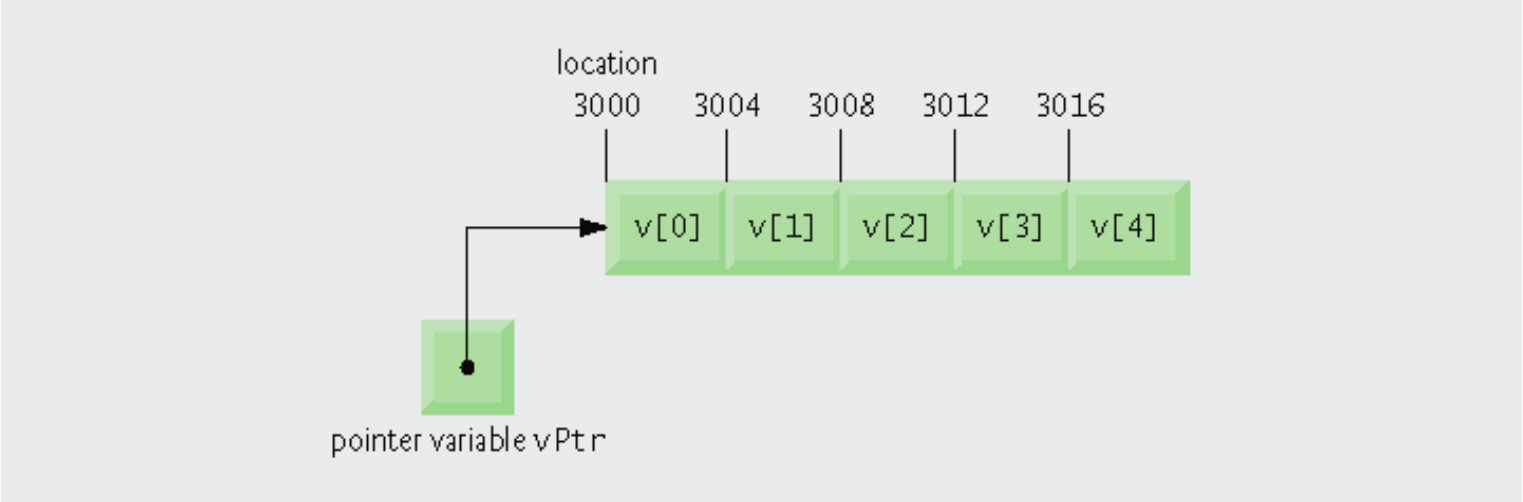


Fig. 8.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

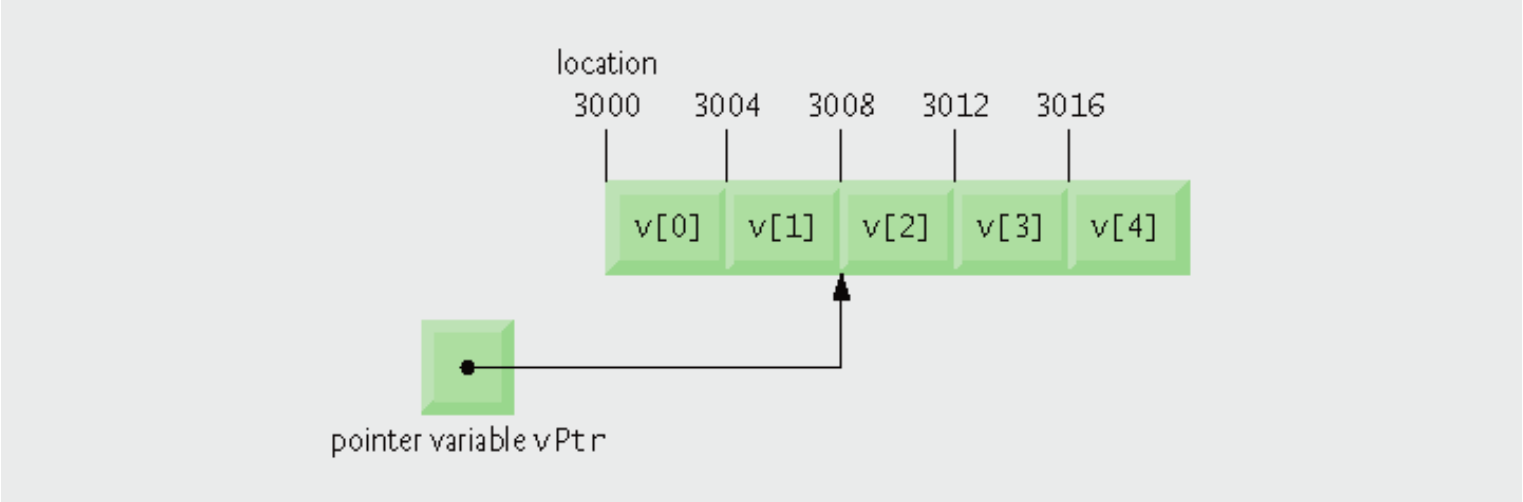


Fig. 8.19 | Pointer `vPtr` after pointer arithmetic.

Common Programming Error 8.9

Using pointer arithmetic on a pointer that does not refer to an array of values is a logic error.

Common Programming Error 8.10

Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.

Common Programming Error 8.11

Using pointer arithmetic to increment or decrement a pointer such that the pointer refers to an element past the end of the array or before the beginning of the array is normally a logic error.

8.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- **Pointer assignment**
  - **Pointer can be assigned to another pointer if both are of same type**
    - **If not same type, cast operator must be used**
    - **Exception**
      - **Pointer to `void` (type `void *`)**
        - **Generic pointer, represents any type**
        - **No casting needed to convert pointer to `void *`**
        - **Casting is needed to convert `void *` to any other type**
        - **`void` pointers cannot be dereferenced**

Software Engineering Observation 8.5

Nonconstant pointer arguments can be passed to constant pointer parameters. This is helpful when the body of a program uses a nonconstant pointer to access data, but does not want that data to be modified by a function called in the body of the program.

Common Programming Error 8.12

Assigning a pointer of one type to a pointer of another (other than `void *`) without casting the first pointer to the type of the second pointer is a compilation error.

Common Programming Error 8.13

All operations on a `void *` pointer are compilation errors, except comparing `void *` pointers with other pointers, casting `void *` pointers to valid pointer types and assigning addresses to `void *` pointers.

8.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- **Pointer comparison**
  - **Use equality and relational operators**
  - **Compare addresses stored in pointers**
    - **Comparisons are meaningless unless pointers point to members of the same array**
  - **Example**
    - **Could show that one pointer points to higher-index element of array than another pointer**
  - **Commonly used to determine whether pointer is 0 (null pointer)**

8.9 Relationship Between Pointers and Arrays

- **Arrays and pointers are closely related**
  - **Array name is like constant pointer**
  - **Pointers can do array subscripting operations**

8.9 Relationship Between Pointers and Arrays (Cont.)

- **Accessing array elements with pointers**
  - **Assume declarations:**

```
int b[ 5 ];
int *bPtr;
bPtr = b;
```

  - **Element `b[ n ]` can be accessed by `*( bPtr + n )`**
    - **Called pointer/offset notation**
  - **Addresses**
    - **`&b[ 3 ]` is same as `bPtr + 3`**
  - **Array name can be treated as pointer**
    - **`b[ 3 ]` is same as `*( b + 3 )`**
  - **Pointers can be subscripted (pointer/subscript notation)**
    - **`bPtr[ 3 ]` is same as `b[ 3 ]`**

Common Programming Error 8.14

Although array names are pointers to the beginning of the array and pointers can be modified in arithmetic expressions, array names cannot be modified in arithmetic expressions, because array names are constant pointers.

Good Programming Practice 8.2

For clarity, use array notation instead of pointer notation when manipulating arrays.

Outline

fig08_20.cpp

(1 of 3)

```
1  // Fig. 8.20: fig08_20.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
10    int *bPtr = b; // set bPtr to point to array b
11
12    // output array b using array subscript notation
13    cout << "Array b printed with:\n\nArray subscript notation\n";
14
15    for ( int i = 0; i < 4; i++ )
16       cout << "b[" << i << "] = " << b[ i ] << '\n';
17
18    // output array b using the array name and pointer/offset notation
19    cout << "\nPointer/offset notation where "
20       << "the pointer is the array name\n";
21
22    for ( int offset1 = 0; offset1 < 4; offset1++ )
23       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Using array subscript notation

Using array name and pointer/offset notation

Outline

fig08_20.cpp

```
24
25    // output array b using bPtr and array subscript notation
26    cout << "\nPointer subscript notation\n";
27
28    for ( int j = 0; j < 4; j++ )
29       cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
30
31    cout << "\nPointer/offset notation\n";
32
33    // output array b using bPtr and pointer/offset notation
34    for ( int offset2 = 0; offset2 < 4; offset2++ )
35       cout << "*(bPtr + " << offset2 << ") = "
36          << *( bPtr + offset2 ) << '\n';
37
38    return 0; // indicates successful termination
39 } // end main
```

Using pointer subscript notation

Using pointer name and 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
```

Outline

fig08_20.cpp

(3 of 3)

Outline

fig08_21.cpp

(1 of 2)

```
1  // Fig. 8.21: fig08_21.cpp
2  // Copying a string using array notation and pointer notation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void copy1( char *, const char * ); // prototype
8  void copy2( char *, const char * ); // prototype
9
10 int main()
11 {
12    char string1[ 10 ];
13    char *string2 = "Hello";
14    char string3[ 10 ];
15    char string4[] = "Good Bye";
16
17    copy1( string1, string2 ); // copy string2 into string1
18    cout << "string1 = " << string1 << endl;
19
20    copy2( string3, string4 ); // copy string4 into string3
21    cout << "string3 = " << string3 << endl;
22    return 0; // indicates successful termination
23 } // end main
```

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

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

Use array subscript notation to copy string in **s2** to character array **s1**

Use pointer notation to copy string in **s2** to character array in **s1**

Increment both pointers to point to next elements in corresponding arrays

fig08_21.cpp

(2 of 2)

8.10 Arrays of Pointers

- **Arrays can contain pointers**
  - **Commonly used to store array of strings (string array)**
    - **Array does not store strings, only pointers to strings**
    - **Example**
      - `const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };`
        - **Each element of `suit` points to a `char *` (string)**
    - **`suit` array has fixed size (4), but strings can be of any size**
    - **Commonly used with command-line arguments to function `main`**

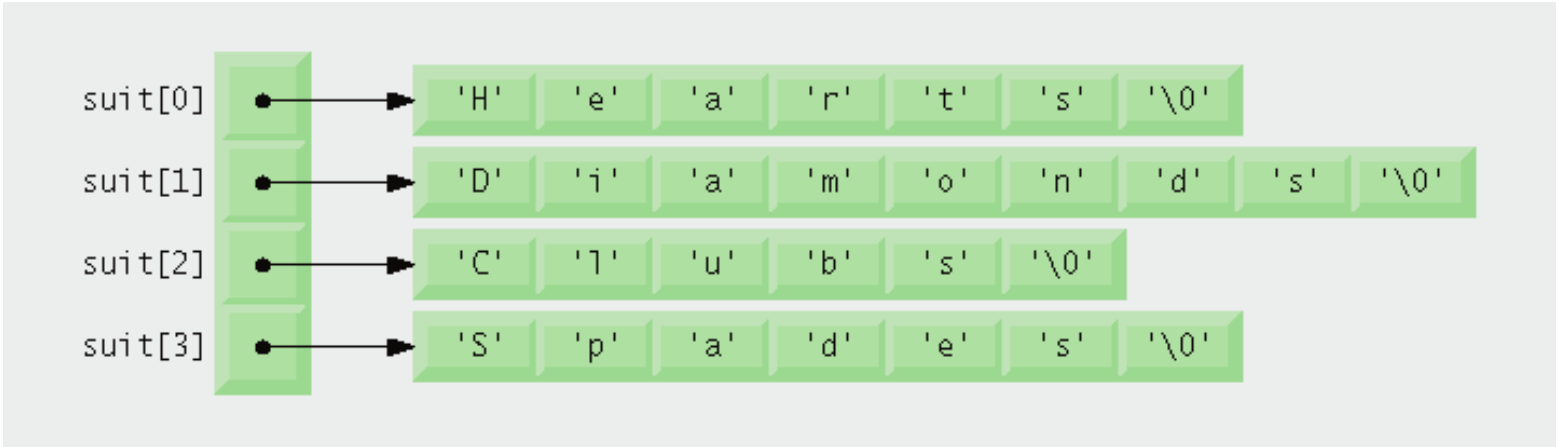


Fig. 8.22 | Graphical representation of the suit array.

8.11 Case Study: Card Shuffling and Dealing Simulation

- **Card shuffling program**
  - **Use an array of pointers to strings, to store suit names**
  - **Use a double scripted array (suit-by-value)**
  - **Place 1-52 into the array to specify the order in which the cards are dealt**
- **Indefinite postponement (starvation)**
  - **An algorithm executing for an indefinitely long period**
    - **Due to randomness**

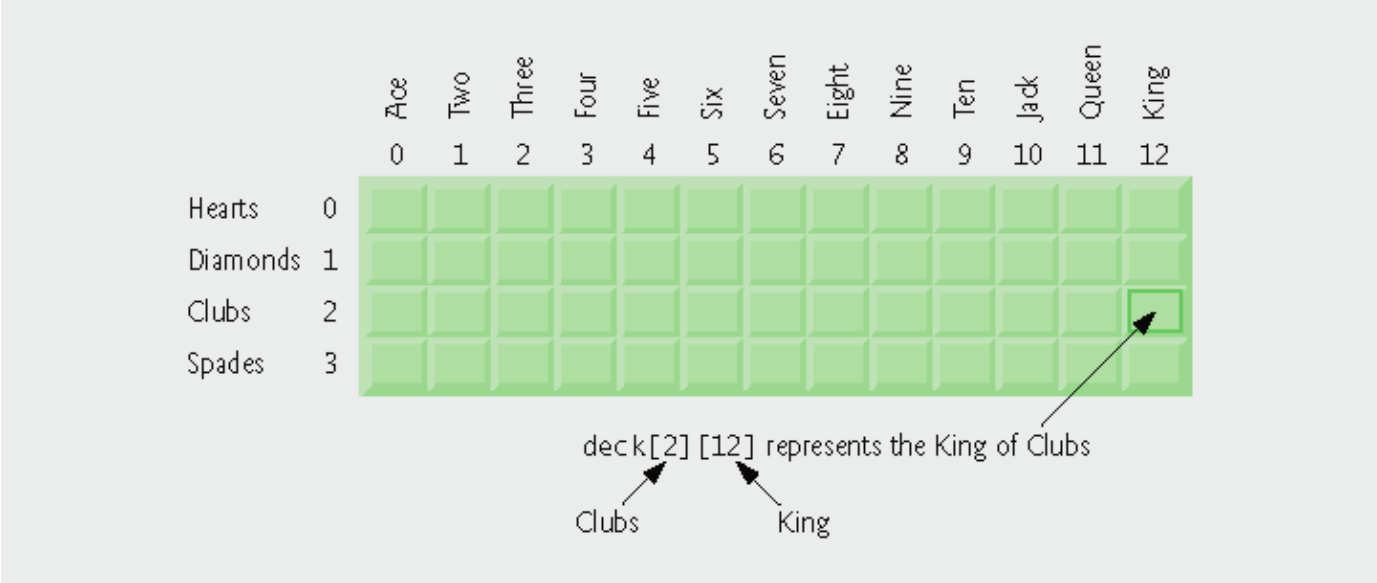


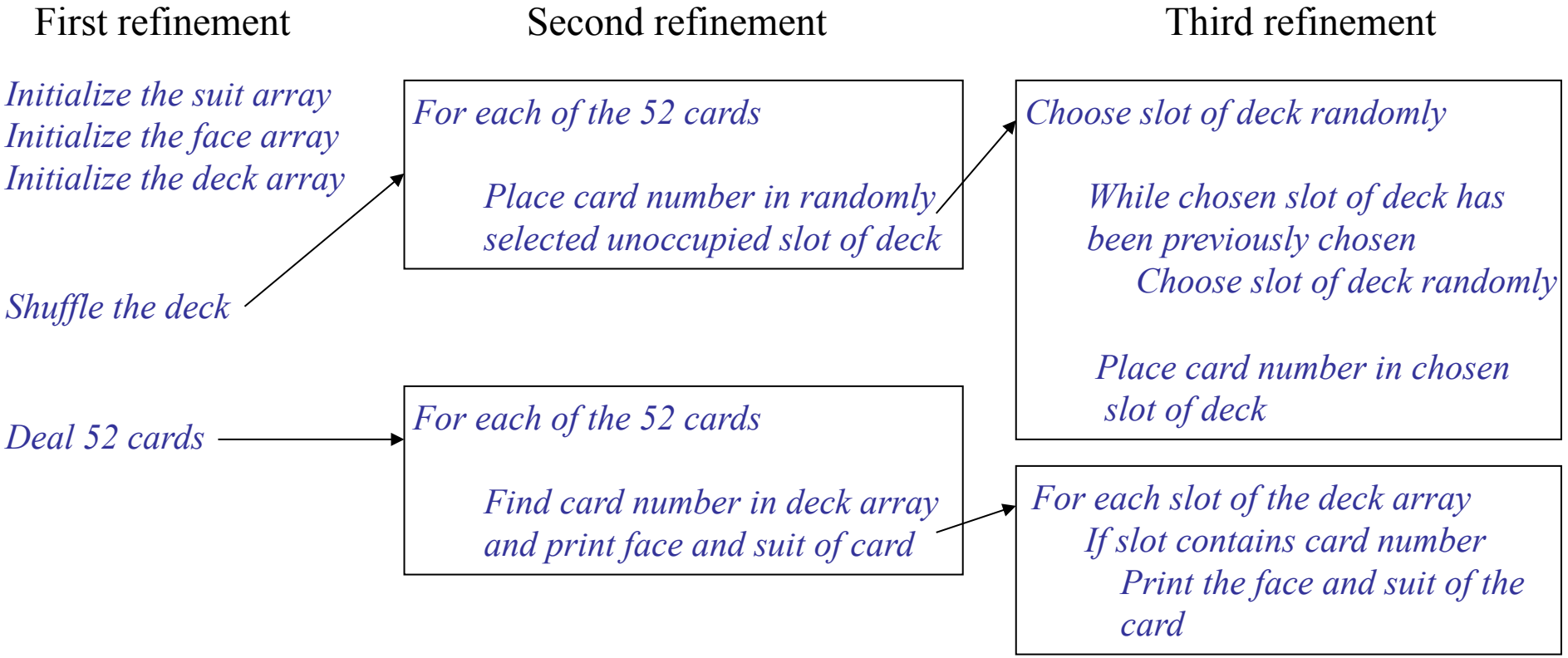
Fig. 8.23 | Two-dimensional array representation of a deck of cards.

Performance Tip 8.3

Sometimes algorithms that emerge in a "natural" way can contain subtle performance problems such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

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

- **Pseudocode for shuffling and dealing simulation**



```
1  Initialize the suit array
2  Initialize the face array
3  Initialize the deck array
4
5  For each of the 52 cards
6     Choose slot of deck randomly
7
8     While slot of deck has been previously chosen
9        Choose slot of deck randomly
10
11    Place card number in chosen slot of deck
12
13 For each of the 52 cards
14    For each slot of deck array
15       If slot contains desired card number
16          Print the face and suit of the card
```

Outline

fig08_24.cpp

(1 of 1)

```
1  // Fig. 8.25: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4
5  // DeckOfCards class definition
6  class DeckOfCards
7  {
8  public:
9     DeckOfCards(); // constructor initializes deck
10    void shuffle(); // shuffles cards in deck
11    void deal(); // deals cards in deck
12 private:
13    int deck[ 4 ][ 13 ]; // represents deck of cards
14 }; // end class DeckOfCards
```

Outline

fig08_25.cpp

(1 of 1)

Outline

fig08_26.cpp

(1 of 4)

```
1  // Fig. 8.26: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  using std::cout;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototypes for rand and srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // prototype for time
17 using std::time;
18
19 #include "DeckOfCards.h" // DeckOfCards class definition
20
```

Outline

fig08_26.cpp

(2 of 4)

```
21 // DeckOfCards default constructor initializes deck
22 DeckOfCards::DeckOfCards()
23 {
24    // loop through rows of deck
25    for ( int row = 0; row <= 3; row++ )
26    {
27       // loop through columns of deck for current row
28       for ( int column = 0; column <= 12; column++ )
29       {
30          deck[ row ][ column ] = 0; // initialize slot of deck to 0
31       } // end inner for
32    } // end outer for
33
34    srand( time( 0 ) ); // seed random number generator
35 } // end DeckOfCards default constructor
36
37 // shuffle cards in deck
38 void DeckOfCards::shuffle()
39 {
40    int row; // represents suit value of card
41    int column; // represents face value of card
42
43    // for each of the 52 cards, choose a slot of the deck randomly
44    for ( int card = 1; card <= 52; card++ )
45    {
46       do // choose a new random location until unoccupied slot
47       {
48          row = rand() % 4; // randomly select the row
49          column = rand() % 13; // randomly select the column
50       } while( deck[ row ][ column ] != 0 ); // end do...while
```

Current position is at randomly selected row and column

Outline

fig08_26.cpp

(3 of 4)

```
51
52       // place card number in chosen slot of deck
53       deck[ row ][ column ] = card;
54    } // end for
55 } // end function shuffle
56
57 // deal cards in deck
58 void DeckOfCards::deal()
59 {
60    // initialize suit array
61    static const char *suit[ 4 ] =
62       { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64    // initialize face array
65    static const char *face[ 13 ] =
66       { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
```

suit array contains pointers to **char** arrays

face array contains pointers to **char** arrays

Outline

fig08_26.cpp

(4 of 4)

```
68
69    // for each of the 52 cards
70    for ( int card = 1; card <= 52; card++ )
71    {
72       // loop through rows of deck
73       for ( int row = 0; row <= 3; row++ )
74       {
75          // loop through columns of deck for current row
76          for ( int column = 0; column <= 12; column++ )
77          {
78             // if slot contains current card, display ca
79             if ( deck[ row ][ column ] == card )
80             {
81                cout << setw( 5 ) << right << face[ column ]
82                   << " of " << setw( 8 ) << left << suit[ row ]
83                   << ( card % 2 == 0 ? '\n' : '\t' );
84             } // end if
85          } // end innermost for
86       } // end inner for
87    } // end outer for
88 } // end function deal
```

Cause face to be output right justified in field of 5 characters

Cause suit to be output left justified in field of 8 characters

Outline

fig08_27.cpp

(1 of 2)

```
1  // Fig. 8.27: fig08_27.cpp
2  // Card shuffling and dealing program.
3  #include "DeckOfCards.h" // DeckOfCards class definition
4
5  int main()
6  {
7     DeckOfCards deckOfCards; // create DeckOfCards object
8
9     deckOfCards.shuffle(); // shuffle the cards in the deck
10    deckOfCards.deal(); // deal the cards in the deck
11    return 0; // indicates successful termination
12 } // end main
```

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

Outline

fig08_27.cpp

(2 of 2)

8.12 Function Pointers

- **Pointers to functions**
  - **Contain addresses of functions**
    - **Similar to how array name is address of first element**
    - **Function name is starting address of code that defines function**
- **Function pointers can be**
  - **Passed to functions**
  - **Returned from functions**
  - **Stored in arrays**
  - **Assigned to other function pointers**

8.12 Function Pointers (Cont.)

- **Calling functions using pointers**
  - **Assume function header parameter:**
    - `bool ( *compare ) ( int, int )`
  - **Execute function from pointer with either**
    - `( *compare ) ( int1, int2 )`
      - **Dereference pointer to function**

    OR
    - `compare( int1, int2 )`
      - **Could be confusing**
        - **User may think `compare` is name of actual function in program**

Outline

fig08_28.cpp

```
1  // Fig. 8.28: fig08_28.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 // prototypes
12 void selectionSort( int [], const int, bool (*)( int, int ) );
13 void swap( int * const, int * const );
14 bool ascending( int, int ); // implements ascending order
15 bool descending( int, int ); // implements descending order
16
17 int main()
18 {
19    const int arraySize = 10;
20    int order; // 1 = ascending, 2 = descending
21    int counter; // array index
22    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24    cout << "Enter 1 to sort in ascending order,\n"
25       << "Enter 2 to sort in descending order: ";
26    cin >> order;
27    cout << "\nData items in original order\n";
```

Parameter is pointer to function that receives two integer parameters and returns **bool** result

Outline

fig08_28.cpp

(2 of 4)

```
28
29    // output original array
30    for ( counter = 0; counter < arraySize; counter++ )
31       cout << setw( 4 ) << a[ counter ];
32
33    // sort array in ascending order; pass function ascending
34    // as an argument to specify ascending sorting order
35    if ( order == 1 )
36    {
37       selectionSort( a, arraySize, ascending );
38       cout << "\nData items in ascending order\n";
39    } // end if
40
41    // sort array in descending order; pass function descending
42    // as an argument to specify descending sorting order
43    else
44    {
45       selectionSort( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    } // end else part of if...else
48
49    // output sorted array
50    for ( counter = 0; counter < arraySize; counter++ )
51       cout << setw( 4 ) << a[ counter ];
52
53    cout << endl;
54    return 0; // indicates successful termination
55 } // end main
```

Pass pointers to functions **ascending** and **descending** as parameters to function **selectionSort**

Outline

```
56
57 // multipurpose selection sort; the parameter compare is a pointer to
58 // the comparison function that determines the sorting c
59 void selectionSort( int work[], const int size,
60                     bool (*compare)( int, int ) )
61 {
62    int smallestOrLargest; // index of smallest (or large
63
64    // loop over size - 1 elements
65    for ( int i = 0; i < size - 1; i++ )
66    {
67       smallestOrLargest = i; // first index of remaining
68
69       // loop to find index of smallest (or largest) eleme
70       for ( int index = i + 1; index < size; index++ )
71          if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
72             smallestOrLargest = index;
73
74       swap( &work[ smallestOrLargest ], &work[ i ] );
75    } // end if
76 } // end function selectionSort
77
78 // swap values at memory locations to which
79 // element1Ptr and element2Ptr point
80 void swap( int * const element1Ptr, int * const element2Ptr )
81 {
82    int hold = *element1Ptr;
83    *element1Ptr = *element2Ptr;
84    *element2Ptr = hold;
85 } // end function swap
```

compare is a pointer to a function that receives two integer parameters and returns a **bool** result

Parentheses necessary to indicate pointer to function

Dereference pointer **compare** to execute the function

8_28.cpp

Outline

fig08_28.cpp

(4 of 4)

```
86
87 // determine whether element a is less than
88 // element b for an ascending order sort
89 bool ascending( int a, int b )
90 {
91    return a < b; // returns true if a is less than b
92 } // end function ascending
93
94 // determine whether element a is greater than
95 // element b for a descending order sort
96 bool descending( int a, int b )
97 {
98    return a > b; // returns true if a is greater than b
99 } // 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
```

8.12 Function Pointers (Cont.)

- **Arrays of pointers to functions**
  - **Menu-driven systems**
    - **Pointers to each function stored in array of pointers to functions**
      - **All functions must have same return type and same parameter types**
    - **Menu choice determines subscript into array of function pointers**

Outline

fig08_29.cpp

(1 of 3)

```
1  // Fig. 8.29: fig08_29.cpp
2  // Demonstrating an array of pointers to functions.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // function prototypes -- each function performs similar actions
9  void function0( int );
10 void function1( int );
11 void function2( int );
12
13 int main()
14 {
15    // initialize array of 3 pointers to functions that each
16    // take an int argument and return void
17    void (*f[ 3 ])( int ) = { function0, function1, function2 };
18
19    int choice;
20
21    cout << "Enter a number between 0 and 2, 3 to end: ";
22    cin >> choice;
```

Array initialized with names of three functions

Outline

fig08_29.cpp

(2 of 3)

```
23
24    // process user's choice
25    while ( ( choice >= 0 ) && ( choice < 3 ) )
26    {
27       // invoke the function at location choice in
28       // the array f and pass choice as an argument
29       (*f[ choice ])( choice );
30
31       cout << "Enter a number between 0 and 2, 3 to end: ";
32       cin >> choice;
33    } // end while
34
35    cout << "Program execution completed." << endl;
36    return 0; // indicates successful termination
37 } // end main
38
39 void function0( int a )
40 {
41    cout << "You entered " << a << " so function0 was called\n\n";
42 } // end function function0
43
44 void function1( int b )
45 {
46    cout << "You entered " << b << " so function1 was called\n\n";
47 } // end function function1
```

Call chosen function by dereferencing corresponding element in array

```
48
49 void function2( int c )
50 {
51    cout << "You entered " << c << " so function2 was called\n\n";
52 } // end function function2
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function1 was called

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

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

Outline

fig08_29.cpp

(3 of 3)

8.13 Introduction to Pointer-Based String Processing

- **Standard Library functions for string processing**
  - **Appropriate for developing text-processing software**

8.13.1 Fundamentals of Characters and Pointer-Based Strings

- **Character constant**
  - **Integer value represented as character in single quotes**
    - **Example**
      - `'z'` **is integer value of** `z`
        - `122` **in ASCII**
      - `'\n'` **is integer value of newline**
        - `10` **in ASCII**

8.13.1 Fundamentals of Characters and Pointer-Based Strings (Cont.)

- **String**
  - Series of characters treated as single unit
  - Can include letters, digits, special characters +, -, *, ...
  - String literal (string constants)
    - Enclosed in double quotes, for example:

      `"I like C++ "`

    - Have `static` storage class
  - Array of characters, ends with null character `'\0'`
  - String is constant pointer
    - Pointer to string's first character
      - Like arrays

8.13.1 Fundamentals of Characters and Pointer-Based Strings (Cont.)

- **String assignment**
  - Character array
    - `char color[] = "blue";`
      - Creates 5 element `char` array `color`
        - Last element is `'\0'`
  - Variable of type `char *`
    - `char *colorPtr = "blue";`
      - Creates pointer `colorPtr` to letter `b` in string `"blue"`
        - `"blue"` somewhere in memory
  - Alternative for character array
    - `char color[] = { 'b', 'l', 'u', 'e', '\0' };`

Common Programming Error 8.15

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.

Common Programming Error 8.16

Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.

Error-Prevention Tip 8.4

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it is to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.

8.13.1 Fundamentals of Characters and Pointer-Based Strings (Cont.)

- **Reading strings**
  - **Assign input to character array `word[ 20 ]`**
    - `cin >> word;`
      - **Reads characters until whitespace or EOF**
  - **String could exceed array size**
    - `cin >> setw( 20 ) >> word;`
      - **Reads only up to 19 characters (space reserved for `'\0'`)**

8.13.1 Fundamentals of Characters and Pointer-Based Strings (Cont.)

- `cin.getline`
  - Read line of text
    - `cin.getline( array, size, delimiter );`
      - Copies input into specified `array` until either
        - One less than `size` is reached
        - `delimiter` character is input
    - Example
      - ```
        char sentence[ 80 ];
        cin.getline( sentence, 80, '\n' );
        ```


        ```

Common Programming Error 8.17

Processing a single character as a `char *` string can lead to a fatal runtime error. A `char *` string is a pointer—probably a respectably large integer. However, a character is a small integer (ASCII values range 0–255). On many systems, dereferencing a `char` value causes an error, because low memory addresses are reserved for special purposes such as operating system interrupt handlers—so "memory access violations" occur.

Common Programming Error 8.18

Passing a string as an argument to a function when a character is expected is a compilation error.

8.13.2 String Manipulation Functions of the String-Handling Library

- String handling library `<cstring>` provides functions to
  - Manipulate string data
  - Compare strings
  - Search strings for characters and other strings
  - Tokenize strings (separate strings into logical pieces)
- Data type `size_t`
  - Defined to be an unsigned integral type
    - Such as `unsigned int` or `unsigned long`
  - In header file `<cstring>`

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 );`	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string `s2` to `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero (usually `-1`) or greater than zero (usually `1`) if `s1` is equal to, less than or greater than `s2`, respectively.

Fig. 8.30 | String-manipulation functions of the string-handling library. (Part 1 of 2)

Function prototype	Function description
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if the `n`-character portion of `s1` is equal to, less than or greater than the corresponding `n`-character portion of `s2`, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text. The string is broken up based on the characters contained in string `s2`. For instance, if we were to break the string `"this:is:a:string"` into tokens based on the character `':'`, the resulting tokens would be `"this"`, `"is"`, `"a"` and `"string"`. Function `strtok` returns only one token at a time, however. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.
`size_t strlen( const char *s );`	Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

Fig. 8.30 | String-manipulation functions of the string-handling library. (Part 2 of 2)

Common Programming Error 8.19

Forgetting to include the `<cstring>` header file when using functions from the string-handling library causes compilation errors.

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Copying strings**
  - `char *strcpy( char *s1, const char *s2 )`
    - Copies second argument into first argument
      - First argument must be large enough to store string and terminating null character
  - `char *strncpy( char *s1, const char *s2, size_t n )`
    - Specifies number of characters to be copied from second argument into first argument
      - Does not necessarily copy terminating null character

Common Programming Error 8.20

When using `strncpy`, the terminating null character of the second argument (a `char *` string) will not be copied if the number of characters specified by `strncpy`'s third argument is not greater than the second argument's length. In that case, a fatal error may occur if the programmer does not manually terminate the resulting `char *` string with a null character.

Outline

fig08_31.cpp

(1 of 2)

```
1  // Fig. 8.31: fig08_31.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcpy and strncpy
8  using std::strcpy;
9  using std::strncpy;
10
11 int main()
12 {
13    char x[] = "Happy Birthday to You"; // string length 21
14    char y[ 25 ];
15    char z[ 15 ];
16
17    strcpy( y, x ); // copy contents of x into y
18
19    cout << "The string in array x is: " << x
20       << "\nThe string in array y is: " << y << '\n';
```

<cstring> contains prototypes for **strcpy** and **strncpy**

Copy entire string in array **x** into array **y**

```
21
22    // copy first 14 characters of x into z
23    strncpy( z, x, 14 ); // does not copy null character
24    z[ 14 ] = '\0'; // append '\0' to z's contents
25
26    cout << "The string in array z is: " << z << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Copy first 14 characters of array **x** into array **y**. Note that this does not write terminating null character

fig08_31.cpp

Append terminating null character

String to copy

Copied string using **strcpy**

Copied first 14 characters using **strncpy**

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Concatenating strings**
  - `char *strcat( char *s1, const char *s2 )`
    - Appends second argument to first argument
      - First character of second argument replaces null character terminating first argument
      - You must ensure first argument large enough to store concatenated result and null character
  - `char *strncat( char *s1, const char *s2, size_t n )`
    - Appends specified number of characters from second argument to first argument
      - Appends terminating null character to result

Outline

fig08_32.cpp

(1 of 2)

```
1  // Fig. 8.32: fig08_32.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcat and strncat
8  using std::strcat;
9  using std::strncat;
10
11 int main()
12 {
13    char s1[ 20 ] = "Happy "; // length 6
14    char s2[] = "New Year "; // length 9
15    char s3[ 40 ] = "";
16
17    cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
20
21    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23    // concatenate first 6 characters of s1 to s3
24    strncat( s3, s1, 6 ); // places '\0' after last character
25
26    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27       << "\ns3 = " << s3;
```

<cstring> contains prototypes for strcat and strncat

Append s2 to s1

Append first 6 characters of s1 to s3

```
28
29    strcat( s3, s1 ); // concatenate s1 to s3
30    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31       << "\ns3 = " << s3 << endl;
32    return 0; // indicates successful termination
33 } // end main
```

Append **s1** to **s3**

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Outline

`fig08_32.cpp`

(2 of 2)

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Comparing strings**
  - `int strcmp( const char *s1, const char *s2 )`
    - Compares character by character
    - Returns
      - Zero if strings are equal
      - Negative value if first string is less than second string
      - Positive value if first string is greater than second string
  - `int strncmp( const char *s1, const char *s2, size_t n )`
    - Compares up to specified number of characters
      - Stops if it reaches null character in one of arguments

Common Programming Error 8.21

Assuming that `strcmp` and `strncmp` return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the `strcmp` or `strncmp` function should be compared with zero to determine whether the strings are equal.

Outline

fig08_33.cpp

(1 of 2)

```
1  // Fig. 8.33: fig08_33.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstring> // prototypes for strcmp and strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16    char *s1 = "Happy New Year";
17    char *s2 = "Happy New Year";
18    char *s3 = "Happy Holidays";
19
20    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
22       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
23       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
24
25    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
26       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
27       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = "
28       << strncmp( s3, s1, 7 ) << endl;
29    return 0; // indicates successful termination
30 } // end main
```

<cstring> contains prototypes for **strcmp** and **strncmp**

Compare **s1** and **s2**

Compare **s1** and **s3**

Compare **s3** and **s1**

Compare up to 6 characters of **s1** and **s3**

Compare up to 7 characters of **s1** and **s3**

Compare up to 7 characters of **s3** and **s1**

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Outline

fig08_33.cpp

(2 of 2)

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Comparing strings (Cont.)**
  - **Characters represented as numeric codes**
    - **Strings compared using numeric codes**
  - **Character codes / character sets**
    - **ASCII**
      - **“American Standard Code for Information Interchage”**
    - **EBCDIC**
      - **“Extended Binary Coded Decimal Interchange Code”**
    - **Unicode**

Portability Tip 8.5

The internal numeric codes used to represent characters may be different on different computers, because these computers may use different character sets.

Portability Tip 8.6

Do not explicitly test for ASCII codes, as in `if ( rating == 65 );` rather, use the corresponding character constant, as in `if ( rating == 'A' )`.

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Tokenizing**
  - **Breaking strings into tokens**
    - **Tokens usually logical units, such as words (separated by spaces)**
    - **Separated by delimiting characters**
  - **Example**
    - **`"This is my string"` has 4 word tokens (separated by spaces)**

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Tokenizing (Cont.)**
  - `char *strtok( char *s1, const char *s2 )`
    - Multiple calls required
      - First call contains two arguments, string to be tokenized and string containing delimiting characters
        - Finds next delimiting character and replaces with null character
      - Subsequent calls continue tokenizing
        - Call with first argument `NULL`
        - Stores pointer to remaining string in a `static` variable
    - Returns pointer to current token

Outline

fig08_34.cpp

(1 of 2)

```
1  // Fig. 8.34: fig08_34.cpp
2  // Using strtok.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototype for strtok
8  using std::strtok;
9
10 int main()
11 {
12    char sentence[] = "This is a sentence with 7 tokens";
13    char *tokenPtr;
14
15    cout << "The string to be tokenized is:\n" << sentence
16       << "\n\nThe tokens are:\n\n";
17
18    // begin tokenization of sentence
19    tokenPtr = strtok( sentence, " " );
20
21    // continue tokenizing sentence until tokenPtr becomes NULL
22    while ( tokenPtr != NULL )
23    {
24       cout << tokenPtr << '\n';
25       tokenPtr = strtok( NULL, " " ); // get next token
26    } // end while
27
28    cout << "\nAfter strtok, sentence = " << sentence << endl;
29    return 0; // indicates successful termination
30 } // end main
```

<cstring> contains prototype for **strtok**

First call to **strtok** begins tokenization

Subsequent calls to **strtok** with **NULL** as first argument to indicate continuation

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

Outline

fig08_34.cpp

(2 of 2)

Common Programming Error 8.22

Not realizing that `strtok` modifies the string being tokenized and then attempting to use that string as if it were the original unmodified string is a logic error.

8.13.2 String Manipulation Functions of the String-Handling Library (Cont.)

- **Determining string lengths**
  - `size_t strlen( const char *s )`
    - **Returns number of characters in string**
      - **Terminating null character is not included in length**
      - **This length is also the index of the terminating null character**

Outline

fig08_35.cpp

(1 of 1)

```
1  // Fig. 8.35: fig08_35.cpp
2  // Using strlen.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototype for strlen
8  using std::strlen;
9
10 int main()
11 {
12    char *string1 = "abcdefghijklmnopqrstuvwxyz";
13    char *string2 = "four";
14    char *string3 = "Boston";
15
16    cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
17       << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
18       << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
19       << endl;
20    return 0; // indicates successful termination
21 } // end main
```

<cstring> contains prototype for **strlen**

Using **strlen** to determine length of strings

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```