EE1003
Introduction to Computer I

Chapter 7
Pointers

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Learning Objectives

In this chapter you'll 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.
- The close relationships between pointers and arrays.
- To use arrays of pointers.
- Basic pointer-based string processing.
- To use pointers to functions.

Outline

7.1 Introduction

- Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures that can grow and shrink, such as linked lists, queues, stacks and trees.
- This chapter explains basic pointer concepts and reinforces the intimate relationship among arrays and pointers.

7.2 Pointer Variable Declarations and Initialization

- A pointer contains the memory address of a variable.
- The variable name directly references a value, and a pointer indirectly references a value.
- Referencing a value through a pointer is called indirection.

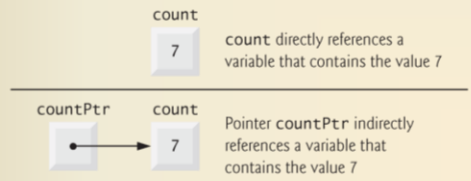


Fig. 7.1 | Directly and indirectly referencing a variable.

7.2 Pointer Variable Declarations and Initialization

- The declaration
  - `int *countPtr, count;`

  declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value)
  - Read as "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - Each variable being declared as a pointer must be preceded by an asterisk (*).
- When * appears in a declaration, it isn't an operator; rather, it indicates that the variable being declared is a pointer.
- Pointers can be declared to point to objects of any data type.

7.2 Pointer Variable Declarations and Initialization

- Pointers should be initialized either when they're declared or in an assignment.
- A pointer may be initialized to 0, `NULL` or an address of the corresponding type.
- A pointer with the value 0 or `NULL` points to nothing and is known as a null pointer.
- The value 0 is the only integer value that can be assigned directly to a pointer variable.

Error-Prevention Tip 7.1
Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

7.3 Pointer Operators

- The address operator (&) is a unary operator that obtains the memory address of its operand.
  - Cannot be applied to constants or to expressions that do not result in references
- Assuming the declarations

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

the following statement assigns the address of the variable `y` to pointer variable `yPtr`.

```
yPtr = &y; // assign address of y to yPtr
```

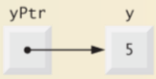


Fig. 7.2 | Graphical representation of a pointer pointing to a variable in memory.

Appendix: Comparing References and Reference Parameters

- References can also be used as aliases for other variables within a function.
- For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for
count
cRef++; // increment count (using its alias cRef)
```

  increments variable `count` by using its alias `cRef`.
- Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.
- Once a reference is declared as an alias for another variable, all operations performed on the alias are actually performed on the original variable.

- Example?

7.3 Pointer Operators

- Assume the integer variable `y` stored at memory location `600000` and pointer variable `yPtr` stored at memory location `500000`.
- The * operator, commonly referred to as the indirection operator or dereferencing operator, returns a synonym for the object to which its pointer operand points.
  - Called dereferencing a pointer
- `yPtr = 600000; *yPtr = 5;`

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

7.3 Pointer Operators

Common Programming Error 7.2

Dereferencing an uninitialized pointer 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 7.4

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

7.3 Pointer Operators

The & and * operators are inverses of one another.

```
1  // Fig. 7.4: fig07_04.cpp
2  // Pointer operators & and *.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int a; // a is an integer
9     int *aPtr; // aPtr is an int * which is a pointer to an integer
10
11    a = 7; // assigned 7 to a
12    aPtr = &a; // assign the address of a to aPtr
13
14    cout << "The address of a is " << &a
15       << "\nThe value of aPtr is " << aPtr;
16    cout << "\n\nThe value of a is " << a
17       << "\nThe value of *aPtr is " << *aPtr;
18    cout << "\n\nShowing that * and & are inverses of "
19       << "each other.\n&*aPtr = " << &*aPtr
20       << "\n*&aPtr = " << *&aPtr << endl;
21 } // end main
```

```
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
```


7.4 Pass-by-Reference with Pointers

- There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.
- In this section, we explain pass-by-reference with pointer arguments.
- Pointers, like references, can be used to modify the variables in the caller or to pass pointers to large data objects to avoid the overhead of being passed by value.
- In C++, you can use pointers and the indirection operator (*) to accomplish pass-by-reference.

7.4 Pass-by-Reference with Pointers

- When calling a function with an argument that should be modified, the address of the argument is passed.
- When the address of a variable is passed to a function, the indirection operator (*) can be used in the function to form a synonym for the name of the variable
  - Used to modify the variable's value at that location in the caller's memory.
- Figure 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`.

7.4 Pass-by-Reference with Pointers

```
1  // Fig. 7.6: fig07_06.cpp
2  // Pass-by-value used to cube a variable's value.
3  #include <iostream>
4  using namespace std;
5
6  int cubeByValue( int ); // prototype
7
8  int main()
9  {
10    int number = 5;
11
12    cout << "The original value of number is " << number;
13
14    number = cubeByValue( number ); // pass number by value to cubeByValue
15    cout << "\nThe new value of number is " << number << endl;
16 } // end main
17
18 // calculate and return cube of integer argument
19 int cubeByValue( int n )
20 {
21    return n * n * n; // cube local variable n and return result
22 } // end function cubeByValue
```

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

7.4 Pass-by-Reference with Pointers

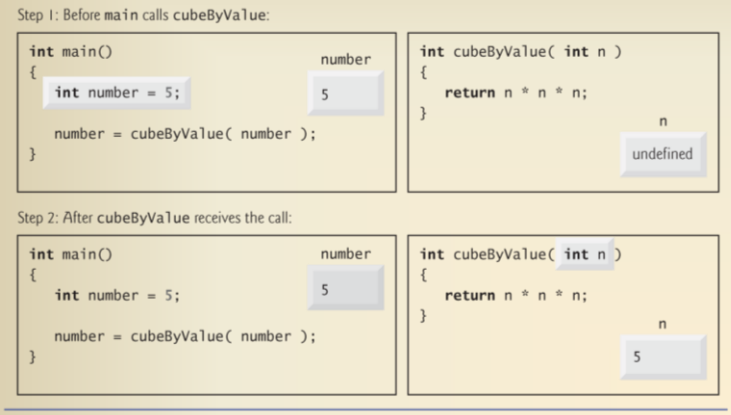
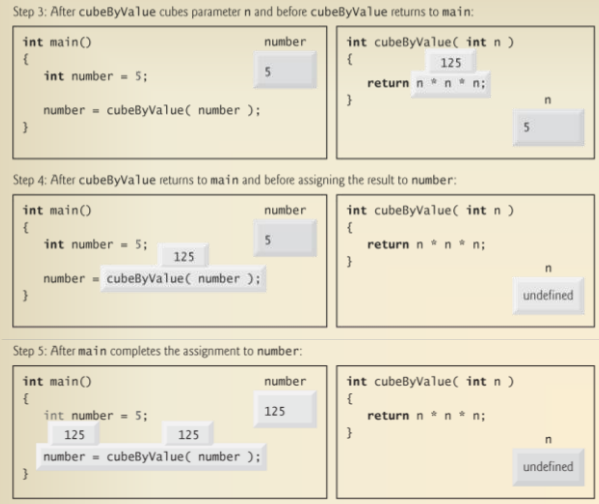


Fig. 7.8 | Pass-by-value analysis of the program of Fig. 7.6. (Part 1 of 3.)

7.4 Pass-by-Reference with Pointers



7.4 Pass-by-Reference with Pointers

```
1  // Fig. 7.7: fig07_07.cpp
2  // Pass-by-reference with a pointer argument used to cube a
3  // variable's value.
4  #include <iostream>
5  using namespace std;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18 } // end main
19
20 // calculate cube of *nPtr; modifies variable number in main
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

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


7.4 Pass-by-Reference with Pointers

- Figure 7.7 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument—the address of `number` is passed to the function.
- The function dereferences the pointer and cubes the value to which `nPtr` points.
  - This directly changes the value of `number` in `main`.
- Figures 7.8–7.9 analyze graphically the execution of the programs in Fig. 7.6 and Fig. 7.7, respectively.

7.4 Pass-by-Reference with Pointers

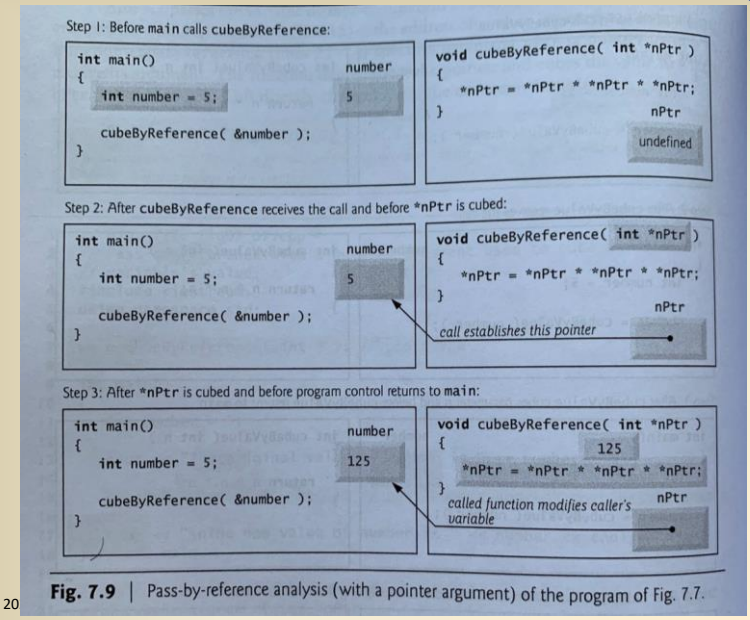


Fig. 7.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 7.7.

Recall

5.15 References and Reference Parameters

```
1  // Fig. 5.18: fig05_18.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue( int ); // function prototype (value pass)
7  void squareByReference( int & ); // function prototype (reference pass)
8
9  int main()
10 {
11    int x = 2; // value to square using squareByValue
12    int z = 4; // value to square using squareByReference
13
14    // demonstrate squareByValue
15    cout << "x = " << x << " before squareByValue\n";
16    cout << "Value returned by squareByValue: "
17       << squareByValue( x ) << endl;
18    cout << "x = " << x << " after squareByValue\n" << endl;
19
20    // demonstrate squareByReference
21    cout << "z = " << z << " before squareByReference" << endl;
22    squareByReference( z );
23    cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 1 of 2.)

Recall

5.15 References and Reference Parameters

```
25
26 // squareByValue multiplies number by itself, stores the
27 // result in number and returns the new value of number
28 int squareByValue( int number )
29 {
30    return number *= number; // caller's argument not modified
31 } // end function squareByValue
32
33 // squareByReference multiplies numberRef by itself and stores the result
34 // in the variable to which numberRef refers in function main
35 void squareByReference( int &numberRef )
36 {
37    numberRef *= numberRef; // caller's argument modified
38 } // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 2 of 2.)

7.5 Using `const` with Pointers

- There are four ways to pass a pointer to a function
  - a nonconstant pointer to nonconstant data
  - a nonconstant pointer to constant data (Fig. 7.10)
  - a constant pointer to nonconstant data (Fig. 7.11)
  - a constant pointer to constant data (Fig. 7.12)
- Each combination provides a different level of access privilege.

Software Engineering Observation 7.2

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared `const`.

7.5 Using `const` with Pointers

- A nonconstant pointer to nonconstant data
  ex: `int *myPtr = &x;`
  - Both the address and the data can be changed
- A nonconstant pointer to constant data (Fig. 7.10)
  ex: `const int *myPtr = &x;`
  - Modifiable pointer to a **`const int`** (data are not modifiable)
- A constant pointer to nonconstant data (Fig. 7.11)
  ex: `int *const myPtr = &x;`
  - Constant pointer to an **`int`** (data can be changed, but the address cannot)
- A constant pointer to constant data (Fig. 7.12)
  ex: `const int *const Ptr = &x;`
  - Both the address and the data are not modifiable

7.5 Using `const` with Pointers

```
1  // Fig. 7.10: fig07_10.cpp
2  // Attempting to modify data through a
3  // nonconstant 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 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17    *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch07\Fig07_10\fig07_10.cpp(17) :
   error C3892: 'xPtr' : you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_10.cpp: In function `void f(const int*)':
fig07_10.cpp:17: error: assignment of read-only location
```

7.5 Using const with Pointers

Performance Tip 7.1

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 7.3

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

7.5 Using const with Pointers

```
1  // Fig. 7.11: fig07_11.cpp
2  // Attempting to modify a constant pointer to nonconstant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch07\Fig07_11\fig07_11.cpp(14) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_11.cpp: In function `int main()':
fig07_11.cpp:14: error: assignment of read-only variable `ptr'
```

7.5 Using const with Pointers

```
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch07\Fig07_12\fig07_12.cpp(17) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
c:\cpphtp7_examples\ch07\Fig07_12\fig07_12.cpp(18) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_12.cpp: In function `int main()':
fig07_12.cpp:17: error: assignment of read-only location
fig07_12.cpp:18: error: assignment of read-only variable `ptr'
```


7.6 Selection Sort Using Pass-by-Reference

- Easy-to-program, but inefficient, sorting algorithm.
- The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element.
- The second iteration selects the smallest element of the remaining elements and swaps it with the second element.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.

7.6 Selection Sort Using Pass-by-Reference

```
1  // Fig. 7.13: fig07_13.cpp
2  // Selection sort with pass-by-reference. This program puts values into an
3  // array, sorts them into ascending order and prints the resulting array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  void selectionSort( int * const, const int ); // prototype
9  void swap( int * const, int * const ); // prototype
10
11 int main()
12 {
13    const int arraySize = 10;
14    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    cout << "Data items in original order\n";
17
18    for ( int i = 0; i < arraySize; i++ )
19       cout << setw( 4 ) << a[ i ];
20
21    selectionSort( a, arraySize ); // sort the array
22
23    cout << "\nData items in ascending order\n";
24
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 1 of 3.)

7.6 Selection Sort Using Pass-by-Reference

```
25    for ( int j = 0; j < arraySize; j++ )
26       cout << setw( 4 ) << a[ j ];
27
28    cout << endl;
29 } // end main
30
31 // function to sort an array
32 void selectionSort( int * const array, const int size )
33 {
34    int smallest; // index of smallest element
35
36    // loop over size - 1 elements
37    for ( int i = 0; i < size - 1; i++ )
38    {
39       smallest = i; // first index of remaining array
40
41       // loop to find index of smallest element
42       for ( int index = i + 1; index < size; index++ )
43
44          if ( array[ index ] < array[ smallest ] )
45             smallest = index;
46
47       swap( &array[ i ], &array[ smallest ] );
48    } // end if
49 } // end function selectionSort
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 2 of 3.)

7.6 Selection Sort Using Pass-by-Reference

```
50
51 // swap values at memory locations to which
52 // element1Ptr and element2Ptr point
53 void swap( int * const element1Ptr, int * const element2Ptr )
54 {
55    int hold = *element1Ptr;
56    *element1Ptr = *element2Ptr;
57    *element2Ptr = hold;
58 } // 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
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 3 of 3.)

7.6 Selection Sort Using Pass-by-Reference

- In the function header and in the prototype for a function that expects a one-dimensional array as an argument, pointer notation may be used.
  - Array name is the pointer to its first element.
- The compiler does not differentiate between a pointer and a one-dimensional array in the function arguments.
  - The function must "know" whether it is an array or simply a pointer to a single variable.
- 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`.
  - Both forms are interchangeable.

7.6 Selection Sort Using Pass-by-Reference

- Because `selectionSort` wants `swap` to have access to the array elements to be swapped, `selectionSort` passes each of these elements to `swap` by reference.
  - The address of each array element is passed explicitly.
- When `swap` references `*element1Ptr`, it's actually referencing `array[ i ]` in `selectionSort`.
- When `swap` references `*element2Ptr`, it's actually referencing `array[ smallest ]` in `selectionSort`.

7.7 `sizeof` Operator

- The unary operator sizeof determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation.
- When applied to the name of an array, the sizeof operator returns the total number of bytes in the array as a value of type size_t.
  - size_t type is a base unsigned integer type of C/C++ language
- When applied to a pointer parameter in a function that receives an array as an argument, the sizeof operator returns the size of the pointer in bytes—not the size of the array.

7.7 sizeof Operator

7.7 sizeof Operator

```
1  // Fig. 7.14: fig07_14.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using namespace std;
6
7  size_t getSize( double * ); // prototype
8
9  int main()
10 {
11    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
12
13    cout << "The number of bytes in the array is " << sizeof( array );
14
15    cout << "\nThe number of bytes returned by getSize is "
16       << getSize( array ) << endl;
17 } // end main
18
19 // return size of ptr
20 size_t getSize( double *ptr )
21 {
22    return sizeof( ptr );
23 } // end function getSize
```

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

Fig. 7.14 | sizeof operator when applied to an array name returns the number of bytes in the array. (Part 2 of 2.)

7.7 sizeof Operator

- The number of elements in an array also can be determined using the results of two `sizeof` operations.
- Consider the following array declaration:
  - `double realArray[ 22 ];`
- To determine the number of elements in the array, the following expression (which is evaluated at compile time) can be used:
  - `sizeof realArray / sizeof( realArray[ 0 ] )`
- The expression determines the number of bytes in array `realArray` and divides that value by the number of bytes used in memory to store the array's first element.

7.7 sizeof Operator

- Operator `sizeof` can be applied to any expression or type name.
- Figure 7.15 uses `sizeof` to calculate the number of bytes used to store most of the standard data types.
- When `sizeof` is applied to a variable name (which is not an array name) or other expression, the number of bytes used to store the specific type of the expression's value is returned.
- The parentheses used with `sizeof` are required only if a type name is supplied as its operand.

7.7 sizeof Operator

```
1  // Fig. 7.15: fig07_15.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char c; // variable of type char
9     short s; // variable of type short
10    int i; // variable of type int
11    long l; // variable of type long
12    float f; // variable of type float
13    double d; // variable of type double
14    long double ld; // variable of type long double
15    int array[ 20 ]; // array of int
16    int *ptr = array; // variable of type int *
17
18    cout << "sizeof c = " << sizeof c
19       << "\tsizeof(char) = " << sizeof( char )
20       << "\nsizeof s = " << sizeof s
21       << "\tsizeof(short) = " << sizeof( short )
22       << "\nsizeof i = " << sizeof i
```

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 1 of 2.)

7.7 sizeof Operator

```
23       << "\tsizeof(int) = " << sizeof( int )
24       << "\nsizeof l = " << sizeof l
25       << "\tsizeof(long) = " << sizeof( long )
26       << "\nsizeof f = " << sizeof f
27       << "\tsizeof(float) = " << sizeof( float )
28       << "\nsizeof d = " << sizeof d
29       << "\tsizeof(double) = " << sizeof( double )
30       << "\nsizeof ld = " << sizeof ld
31       << "\tsizeof(long double) = " << sizeof( long double )
32       << "\nsizeof array = " << sizeof array
33       << "\nsizeof ptr = " << sizeof ptr << endl;
34 } // end main
```

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

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 2 of 2.)

7.7 `sizeof` Operator

Portability Tip 7.2

The number of bytes used to store a particular data type may vary among 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.*

Error-Prevention Tip 7.3

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

7.8 Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- pointer arithmetic—certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type

7.8 Pointer Expressions and Pointer Arithmetic

Portability Tip 7.3

Most computers today have four-byte or 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.

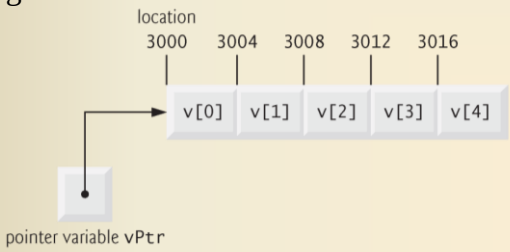
The size of a "long" integer varies between architectures and operating systems.

The Intel® Compiler is compatible and inter-operable with Microsoft* Visual C++ on Windows* and with gcc* on Linux* and Mac OS X*. Consequently, the sizes of fundamental types are the same as for these compilers. The size of a "long" integer in particular depends on the operating system and the targeted architecture as follows:

OS	Architecture	Size of "long" type
Windows	IA-32	4 bytes
	Intel® 64	4 bytes
Linux	IA-32	4 bytes
	Intel® 64	8 bytes
mac OS	Intel® 64	8 bytes

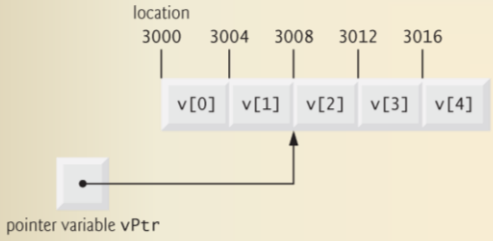
7.8 Pointer Expressions and Pointer Arithmetic

- Assume that array `int v[5]` has been declared and that its first element is at memory location 3000.
- Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000).
- Figure 7.16 diagrams this situation for a machine with four-byte integers.



7.8 Pointer Expressions and Pointer Arithmetic

- In conventional arithmetic, the addition 3000 + 2 yields the value 3002.
  - Not true for pointer arithmetic
- When an integer is added to, or subtracted from, a pointer, the pointer is incremented or decremented 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.



7.8 Pointer Expressions and Pointer Arithmetic

Common Programming Error 7.9
Using pointer arithmetic on a pointer that does not refer to an array is a logic error.

Common Programming Error 7.10
Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.

Common Programming Error 7.11
Using pointer arithmetic to move a pointer outside the bounds of an array is a logic error.

7.8 Pointer Expressions and Pointer Arithmetic

- Pointer variables pointing to the same array may be subtracted from one another.
- For example, if `vPtr` contains the address `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`.
- Pointer arithmetic is meaningless unless performed on a pointer that points to an array.

7.8 Pointer Expressions and Pointer Arithmetic

- A pointer can be assigned to another pointer if both pointers are of the same type.
- Otherwise, a cast operator must be used to convert the pointer to the pointer type on the left of the assignment.
- All pointer types can be assigned to a pointer of type `void *` without casting.
- A `void *` pointer cannot be dereferenced.
  - The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer
  - For a pointer to `void`, this number of bytes cannot be determined from the type.

7.8 Pointer Expressions and Pointer Arithmetic

- Pointers can be compared using equality and relational operators.
  - Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
  - Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).

7.9 Relationship Between Pointers and Arrays

- The array name (without a subscript) is a **constant** pointer to the first element of the array.
  - Although it's a pointer, it cannot be modified in arithmetic expressions.
- Pointers can be used to do any operation involving array subscripting.
- Assume the following declarations:

```
int b[ 5 ]; // create 5-element int array b
int *bPtr; // create int pointer bPtr
```

- We can set `bPtr` to the address of the first element in array `b` with the statement

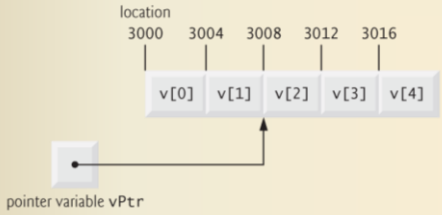
```
bPtr = b; // assign address of array b to bPtr
```

- equivalent to

```
bPtr = &b[ 0 ]; // also assigns address of array
b to bPtr
```

7.9 Relationship Between Pointers and Arrays

- Array element `b[ 2 ]` can alternatively be referenced with the pointer expression
  - `*( bPtr + 2 )`
- The `2` in the preceding expression is the offset to the pointer.
- This notation is referred to as pointer/offset notation.
  - The parentheses are necessary, because the precedence of `*` is higher than that of `+`.



7.9 Relationship Between Pointers and Arrays

- 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 name (which is implicitly `const`) can be treated as a pointer and used in pointer arithmetic.
- For example, the expression `*( b + 3 )` also refers to the array element `b[ 3 ]`.
- In general, all subscripted array expressions can be written with a pointer and an offset.
  - For clarity, use array notation instead of pointer operation when manipulating arrays.

7.9 Relationship Between Pointers and Arrays

- Pointers can be subscripted exactly as arrays can.
- For example, the expression `bPtr[ 1 ]` refers to the array element `b[ 1 ]`; this expression uses pointer/subscript notation.
- In this section, four notations are discussed for referring to array elements to accomplish the same task :
  - Array subscript notation,
  - Pointer/offset notation with the array name as a pointer,
  - Pointer subscript notation, and
  - Pointer/offset notation with a pointer

7.9 Relationship Between Pointers and Arrays

```
1  // Fig. 7.18: fig07_18.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
9     int *bPtr = b; // set bPtr to point to array b
10
11    // output array b using array subscript notation
12    cout << "Array b printed with:\n\nArray subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // output array b using the array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19       << "the pointer is the array name\n";
20
21    for ( int offset1 = 0; offset1 < 4; offset1++ )
22       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)

7.9 Relationship Between Pointers and Arrays

```
23
24      // output array b using bPtr and array subscript notation
25      cout << "\nPointer subscript notation\n";
26
27      for ( int j = 0; j < 4; j++ )
28         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30      cout << "\nPointer/offset notation\n";
31
32      // output array b using bPtr and pointer/offset notation
33      for ( int offset2 = 0; offset2 < 4; offset2++ )
34         cout << "*(bPtr + " << offset2 << ") = "
35            << *( bPtr + offset2 ) << '\n';
36   } // end main
```

```
Array b printed with:

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

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)

7.9 Relationship Between Pointers and Arrays

```
Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

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

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 3 of 3.)

Supplement: Pointer That Points to the Whole Array

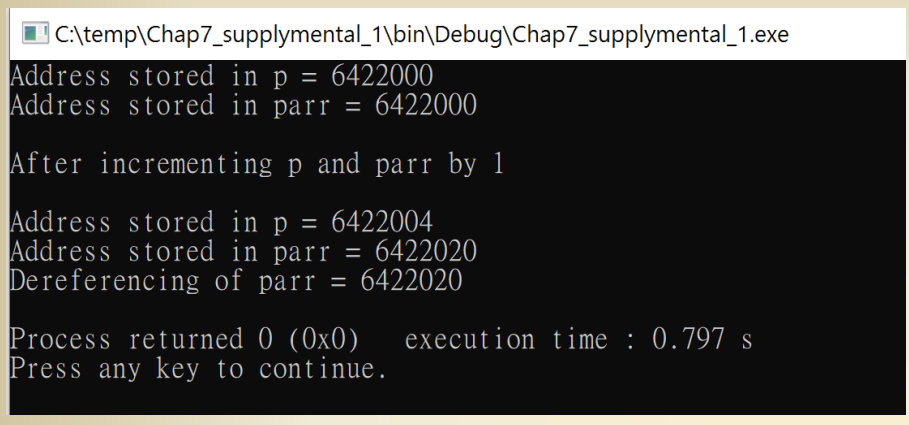
- We have created a pointer which points to the 0th element of the array `int my_arr[5];`
  - `int *p=my_arr;`
  - The base type is (int *) or pointer to int.
- We can also create a pointer that can point to the whole array instead of only one element of the array.
  - `int (*p)[10];`
  - Here p is a pointer that can point to an array of 10 integers
  - The type or base type of p is a pointer to an array of 10 integers.
  - Note that parentheses around p are necessary
- A pointer that points to the 0th element of an array and a pointer that points to the whole array are totally different.

Source: https://overiq.com/c-programming-101/pointers-and-2-d-arrays/

Supplement: Pointer That Points to the Whole Array

```
1  #include <iostream>
2
3  using namespace std;
4
5
6  int main()
7  {
8      int *p; // pointer to int
9      int (*parr)[5]; // pointer to an array of 5 integers
10     int my_arr[5]; // an array of 5 integers
11
12     p = my_arr;
13     parr = &my_arr;
14
15     cout<<"Address stored in p = "<<dec<<(long long)p<<"\n";
16     cout<<"Address stored in parr = "<<(long long)parr<<"\n";
17
18
19     p++;
20     parr++;
21
22     cout<<"\nAfter incrementing p and parr by 1 \n\n";
23     cout<<"Address stored in p = "<<(long long)p<<"\n";
24     cout<<"Address stored in parr = "<<(long long)parr<<"\n";
25     cout<<"Dereferencing of parr = "<<(long long)*parr<<"\n";
26
27     // signal to operating system program ran fine
28     return 0;
29 }
```

Supplement: Pointer That Points to the Whole Array



Supplement: Pointer That Points to the Whole Array

- The pointer arithmetic is performed relative to the base type of the pointer
  - parr is incremented by 20 bytes i.e (5 x 4 = 20 bytes)
  - p is incremented by 4 bytes only
- The important point you need to remember about pointer to an array is
  - Whenever a pointer to an array is dereferenced we get the address (or base address) of the array to which it points
- On dereferencing parr, we get *parr
  - Although parr and *parr points to the same address, but parr's base type is a pointer to an array of 5 integers, while *parr base type is a pointer to int.

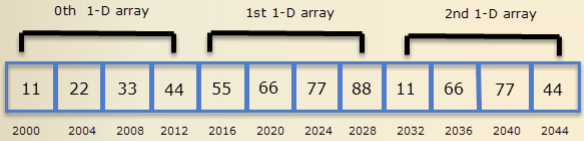
Supplement: Pointers and 2-D Array

- How 2D array looks like

```
int arr[3][4] = {
                    {11,22,33,44},
                    {55,66,77,88},
                    {11,66,77,44}
                };
```

	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

- How it actually store in memory

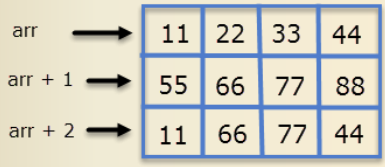


Supplement: Pointers and 2-D Array

- The most important concept you need to remember about a multi-dimensional arrays
  - A 2-D array is actually a 1-D array in which each element is itself a 1-D array. So arr is an array of 3 elements where each element is a 1-D array of 4 integers
- The name of a 1-D array is a constant pointer to the 0^{th} element
  - In a 2-D array, 0^{th} element is a 1-D array

Supplement: Pointers and 2-D Array

- In the above example, the type or base type of arr is a pointer to an array of 4 integers.
  - if arr points to address 2000 then (arr + 1) points to address 2016 (i.e 2000 + 4*4)
  - similarly (arr + 2) will represent the address 2032



Supplement: Pointers and 2-D Array

- In general, we can write (arr + i) points to i^{th} 1-D array
- Dereferencing a pointer to an array gives the base address of the array
  - Dereferencing arr we will get *arr, base type of *arr is (int*)
  - Dereferencing (arr+1) we will get *(arr+1)
  - In general, *(arr+i) points to the base address of the i^{th} 1-D array.
- Again it is important to note that type (arr + i) and *(arr+i) points to same address but their base types are completely different
  - The base type of (arr + i) is a pointer to an array of 4 integers
  - the base type of *(arr + i) is a pointer to int or (int*)

Supplement: Pointers and 2-D Array

- So how you can use arr to access individual elements of a 2-D array?
- Since *(arr + i) points to the base address of every ith 1-D array and it is of base type pointer to int, by using pointer arithmetic we should we able to access elements of ith 1-D array
  - *(arr + i) points to the address of the 0th element of the 1-D array
  - *(arr + i) + 1 points to the address of the 1st element of the 1-D array
  - *(arr + i) + 2 points to the address of the 2nd element of the 1-D array
  - *(arr + i) + j points to the base address of jth element of ith 1-D array
- On dereferencing *(arr + i) + j we will get the value of jth element of ith 1-D array.
  - *(*(arr + i) + j)

Supplement: Pointers and 2-D Array

```
#include <iostream>

using namespace std;


int main()
{
    int arr[3][4] = {
                        {11,22,33,44},
                        {55,66,77,88},
                        {11,66,77,44}
                    };

    int i, j;

    for(i = 0; i < 3; i++)
    {
        cout << "Address of " << i << " th array " << (long long)*(arr + i) << endl;
        for(j = 0; j < 4; j++)
        {
            cout << "arr[" << i << "][" << j << "]=" << *( *(arr + i) + j) << endl;
        }
        cout <<"\n\n" ;
    }

    // signal to operating system program ran fine
    return 0;
}
```

Supplement: Pointers and 2-D Array

```
Address of 0 th array 6421984
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44


Address of 1 th array 6422000
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88


Address of 2 th array 6422016
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44



Process returned 0 (0x0)   execution time : 0.055 s
Press any key to continue.
```


7.10 Pointer-Based String Processing

- This section introduces C-style, pointer-based strings.
- C++'s string class is preferred for use in new programs, because it eliminates many of the security problems that can be caused by manipulating C strings.
- We cover C strings here for a deeper understanding of arrays.
- Also, if you work with legacy C++ programs, you may be required to manipulate these pointer-based strings.

7.10 Pointer-Based String Processing

- Character constant
  - An integer value represented as a character in single quotes.
  - The value of a character constant is the integer value of the character in the machine's character set.
- A string is a series of characters treated as a single unit.
  - May include letters, digits and various special characters such as +, –, *, /and $.
- String literals, or string constants, in C++ are written in double quotation marks
- A pointer-based string is an array of characters ending with a null character (\0).
  - Ex: "happy" → 'h', 'a', 'p', 'p', 'y', '\0'

7.10 Pointer-Based String Processing

Common Programming Error 7.15

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

Error-Prevention Tip 7.4

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's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.

7.10 Pointer-Based String Processing

- The `sizeof` a string literal is the length of the string *including the terminating null character.*
- A string literal may be used as an initializer in the declaration of either a character array or a variable of type `char *`.
  - Ex: char *str1 = “happy” ;
- String literals have `static` storage class (they exist for the duration of the program).
- Modifying a string literal is **undefined**; thus, you should always declare a pointer to a string literal as `const char *`.
- When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character.

7.10 Pointer-Based String Processing

- Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation.
- A string can be read into a character array using stream extraction with `cin`.
- The `setw` stream manipulator can be used to ensure that the read string does not exceed the size of the array.
  - Applies only to the next value being input.

Common Programming Error 7.17

Not providing `cin >>` with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other serious runtime errors.

7.10 Pointer-Based String Processing

- In some cases, it's desirable to input an entire line of text into a character array.
  - The `cin` object provides the member function **getline**.
- Three arguments—a character array in which the line of text will be stored, a length and a delimiter character.
  - Ex: cin.getline(str, 80);
- The function stops reading characters when
  - the delimiter character `'\n'` is encountered, or
  - the end-of-file indicator is entered, or
  - the number of characters read so far is one less than the length specified in the second argument.
- The default value of the third argument is `'\n'`.

7.10 Pointer-Based String Processing

Common Programming Error 7.18

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 from 0 to 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 7.19

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

7.10 Pointer-Based String Processing

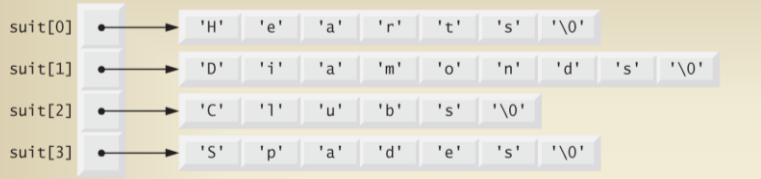
- A character array representing a null-terminated string can be output with `cout` and <<.
- The characters of the string are output until a terminating null character is encountered.
  - The null character is not printed.
- `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters.
  - They do not provide similar input and output processing capabilities for other array types.

7.11 Arrays of Pointers

- Arrays may contain pointers.
- A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a string array.
- Each entry in the array is a string → each entry in an array of strings is simply a pointer to the first character of a string.
- ```
  const char * const suit[ 4 ] =
      { "Hearts", "Diamonds",
        "Clubs", "Spades" };
  ```
  - An array of four elements.
  - Each element is of type "pointer to `char` constant data."


  ```

7.11 Arrays of Pointers



cout << suit[0]; → print out "Hearts"

cout << suit[2]; → print out "Clubs"

What will be printed by the following program?

```
for (int i=0; i<10; i++) {
        idx = rand()%4;
        cout << suit[idx];
}
```

Hearts → Spades → Diamonds → Clubs, … (random strings)

7.12 Function Pointers

- A pointer to a function contains the function's address in memory.
- A function's name is actually the starting address in memory of the code that performs the function's task.
- Pointers to functions can be
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers
  - Used to call the underlying function

7.12 Function Pointers

- To illustrate the use of pointers to functions, Fig. 7.20 modifies the selection sort program of Fig. 7.13.
- Function `selectionSort` receives a pointer to a function—either function `ascending` or function `descending`—as an argument in addition to the integer array to sort and the size of the array.
- Functions `ascending` and `descending` determine the sorting order.

7.12 Function Pointers

```
1  // Fig. 7.20: fig07_20.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // prototypes
8  void selectionSort( int [], const int, bool (*)( int, int ) );
9  void swap( int * const, int * const );
10 bool ascending( int, int ); // implements ascending order
11 bool descending( int, int ); // implements descending order
12
13 int main()
14 {
15    const int arraySize = 10;
16    int order; // 1 = ascending, 2 = descending
17    int counter; // array index
18    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19
20    cout << "Enter 1 to sort in ascending order,\n"
21       << "Enter 2 to sort in descending order: ";
22    cin >> order;
23    cout << "\nData items in original order\n";
24
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 1 of 5.)

7.12 Function Pointers

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

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 2 of 5.)

7.12 Function Pointers

```
49     cout << endl;
50  } // end main
51
52  // multipurpose selection sort; the parameter compare is a pointer to
53  // the comparison function that determines the sorting order
54  void selectionSort( int work[], const int size,
55                      bool (*compare)( int, int ) )
56  {
57     int smallestOrLargest; // index of smallest (or largest) element
58
59     // loop over size - 1 elements
60     for ( int i = 0; i < size - 1; i++ )
61     {
62        smallestOrLargest = i; // first index of remaining vector
63
64        // loop to find index of smallest (or largest) element
65        for ( int index = i + 1; index < size; index++ )
66           if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
67              smallestOrLargest = index;
68
69        swap( &work[ smallestOrLargest ], &work[ i ] );
70     } // end if
71  } // end function selectionSort
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 3 of 5.)

7.12 Function Pointers

```
72
73 // swap values at memory locations to which
74 // element1Ptr and element2Ptr point
75 void swap( int * const element1Ptr, int * const element2Ptr )
76 {
77    int hold = *element1Ptr;
78    *element1Ptr = *element2Ptr;
79    *element2Ptr = hold;
80 } // end function swap
81
82 // determine whether element a is less than
83 // element b for an ascending order sort
84 bool ascending( int a, int b )
85 {
86    return a < b; // returns true if a is less than b
87 } // end function ascending
88
89 // determine whether element a is greater than
90 // element b for a descending order sort
91 bool descending( int a, int b )
92 {
93    return a > b; // returns true if a is greater than b
94 } // end function descending
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 4 of 5.)

7.12 Function Pointers

```
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
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 5 of 5.)

7.12 Function Pointers

- `bool ( *compare )( int, int )`
- This parameter specifies a pointer to a function.
  - The keyword `bool` indicates that the function being pointed to returns a `bool` value.
  - The text `(*compare)` indicates the name of the pointer to the function (the `*` indicates that parameter `compare` is a pointer).
  - The text `(int, int)` indicates that the function pointed to by `compare` takes two integer arguments.
  - Parentheses are needed around `*compare` to indicate that `compare` is a pointer to a function.
- The corresponding parameter in the function prototype of `selectionSort` is
  - `bool (*)( int, int )`

7.12 Function Pointers

- The function passed to `selectionSort` is called as follows:
  - `( *compare )( work[ smallestOrLargest ], work[ index ] )`
- Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to execute the function.
- The parentheses around `*compare` are necessary; otherwise, the `*` operator would attempt to dereference the value returned from the function call.
- Call could have been made without dereferencing the pointer, as in
  - `compare( work[ smallestOrLargest ], work[ index ] )`
- which uses the pointer directly as the function name.

Summary

- Concept of Pointer
- Pointer Declaration
- Pointer Operators
- Pass-by-reference with Pointers
- Pointer Expression and Arithhmetic
- String
- Array
- Function Pointer