Chapter 8
Pointers

C++ How to Program, 9/e

OBJECTIVES

In this chapter you'll:

- Learn what pointers are.
- Learn the similarities and differences between pointers and references.
- Use pointers to pass arguments to functions by reference.
- Understand the close relationships between pointers and built-in arrays.
- Use pointer-based strings.
- Use built-in arrays.
- Use C++11 capabilities, including `nullptr` and Standard Library functions `begin` and `end`.

8.1 Introduction
8.2 Pointer Variable Declarations and Initialization
8.3 Pointer Operators
8.4 Pass-by-Reference with Pointers
8.5 Built-In Arrays
8.6 Using `const` with Pointers
- 8.6.1 Nonconstant Pointer to Nonconstant Data
- 8.6.2 Nonconstant Pointer to Constant Data
- 8.6.3 Constant Pointer to Nonconstant Data
- 8.6.4 Constant Pointer to Constant Data

8.7 `sizeof` Operator
8.8 Pointer Expressions and Pointer Arithmetic
8.9 Relationship Between Pointers and Built-In Arrays
8.10 Pointer-Based Strings
8.11 Wrap-Up

8.1 Introduction

- Pointers are one of the most powerful, yet challenging to use, C++ capabilities.
- Our goals here are to help you determine when it's appropriate to use pointers, and show how to use them *correctly* and *responsibly*.
- 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.
- We also show the intimate relationship among *built-in arrays* and pointers.
- *In new software development projects, you should favor* `array` *and* `vector` *objects to built-in arrays.*

8.2 Pointer Variable Declarations and Initialization

Indirection

- A pointer contains the *memory address* of a variable that, in turn, contains a specific value.
- In this sense, a variable name directly references a value, and a pointer indirectly references a value.
- Referencing a value through a pointer is called indirection.
- Diagrams typically represent a pointer as an *arrow* from the *variable that contains an address to the variable located at that address* in memory.

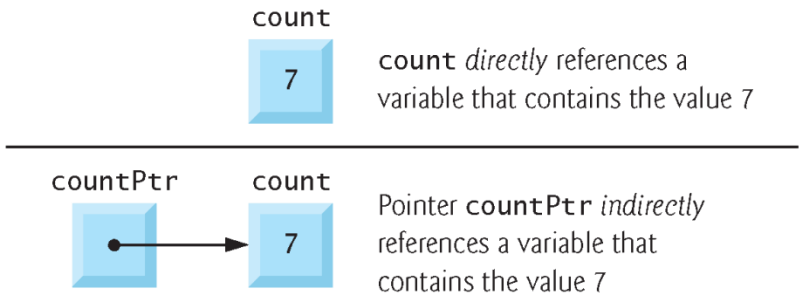

Fig. 8.1 | Directly and indirectly referencing a variable.

8.2 Pointer Variable Declarations and Initialization (cont.)

Declaring Pointers

- The declaration

```
int *countPtr, count;
```

  declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read (right to left), "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - The * in the declaration applies only to `countPtr`.
  - Each variable being declared as a pointer must be preceded by an asterisk (*).
- When * appears in a declaration, it is *not* 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.

Common Programming Error 8.1

Assuming that the * used to declare a pointer distributes to all 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 (with or without spaces in between). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.

Good Programming Practice 8.1

Although it's not a requirement, including the letters `Ptr` in a pointer variable name makes it clear that the variable is a pointer and that it must be handled accordingly.

8.2 Pointer Variable Declarations and Initialization (cont.)

Initializing Pointers

- Pointers should be initialized to `nullptr` (new in C++11) or an address of the corresponding type either when they're declared or in an assignment.
- A pointer with the value `nullptr` "points to nothing" and is known as a null pointer.
- From this point forward, when we refer to a "null pointer" we mean a pointer with the value `nullptr`.

Error-Prevention Tip 8.1

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

8.2 Pointer Variable Declarations and Initialization (cont.)

Null Pointers Prior to C++11

- In earlier versions of C++, the value specified for a null pointer was `0` or `NULL`.
- `NULL` is defined in several standard library headers to represent the value `0`.
- Initializing a pointer to `NULL` is equivalent to initializing a pointer to `0`, but prior to C++11, `0` was used by convention.
- The value `0` is the *only* integer value that can be assigned directly to a pointer variable without first *casting* the integer to a pointer type.

8.3 Pointer Operators

Address (&) Operator

- The address operator (`&`) is a unary operator that *obtains the memory address of its operand*.
- Assuming the declarations

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

the statement

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

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

- Figure 8.2 shows a representation of memory after the preceding assignment.

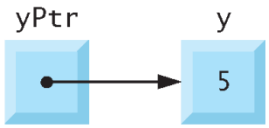

Fig. 8.2 | Graphical representation of a pointer pointing to a variable in memory.

8.3 Pointer Operators (cont.)

- Figure 8.3 shows another pointer representation in memory with integer variable `y` stored at memory location 600000 and pointer variable `yPtr` stored at location 500000.
- The operand of the address operator must be an *lvalue*—the address operator *cannot* be applied to constants or to expressions that result in temporary values (like the results of calculations).