

# Pointers

Chapter 8 of C++ How to Program, 10/e

## OBJECTIVES

In this chapter you'll:

- Learn what pointers are.
- Declare and initialize pointers.
- Use the address (`&`) and indirection (`*`) pointer operators.
- Learn the similarities and differences between pointers and references.
- Use pointers to pass arguments to functions by reference.
- Use built-in arrays.
- Use `const` with pointers.
- Use operator `sizeof` to determine the number of bytes that store a value of a particular type.
- Understand pointer expressions and pointer arithmetic.
- Understand the close relationships between pointers and built-in arrays.
- Use pointer-based strings.
- Use C++11 capabilities, including `nullptr` and Standard Library functions `begin` and `end`.

---

## 8.1 Introduction

## 8.2 Pointer Variable Declarations and Initialization

- 8.2.1 Declaring Pointers
- 8.2.2 Initializing Pointers
- 8.2.3 Null Pointers Prior to C++11

## 8.3 Pointer Operators

- 8.3.1 Address (`&`) Operator
- 8.3.2 Indirection (`*`) Operator
- 8.3.3 Using the Address (`&`) and Indirection (`*`) Operators

## 8.4 Pass-by-Reference with Pointers

## 8.5 Built-In Arrays

- 8.5.1 Declaring and Accessing a Built-In Array
  - 8.5.2 Initializing Built-In Arrays
  - 8.5.3 Passing Built-In Arrays to Functions
  - 8.5.4 Declaring Built-In Array Parameters
  - 8.5.5 C++11: Standard Library Functions `begin` and `end`
  - 8.5.6 Built-In Array Limitations
  - 8.5.7 Built-In Arrays Sometimes Are Required
-

## **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

- 8.8.1 Adding Integers to and Subtracting Integers from Pointers
- 8.8.2 Subtracting Pointers
- 8.8.3 Pointer Assignment
- 8.8.4 Cannot Dereference a `void*`
- 8.8.5 Comparing Pointers

## **8.9** Relationship Between Pointers and Built-In Arrays

- 8.9.1 Pointer/Offset Notation
- 8.9.2 Pointer/Offset Notation with the Built-In Array's Name as the Pointer
- 8.9.3 Pointer/Subscript Notation
- 8.9.4 Demonstrating the Relationship Between Pointers and Built-In Arrays

## **8.10** Pointer-Based Strings (Optional)

## **8.11** Note About Smart Pointers

## **8.12** Wrap-Up

## 8.1 Introduction

- ▶ Pointers are one of the most powerful, yet challenging to use, C++ capabilities.
- ▶ We'll discuss 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 pointer-based dynamic data structures.
- ▶ 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.*

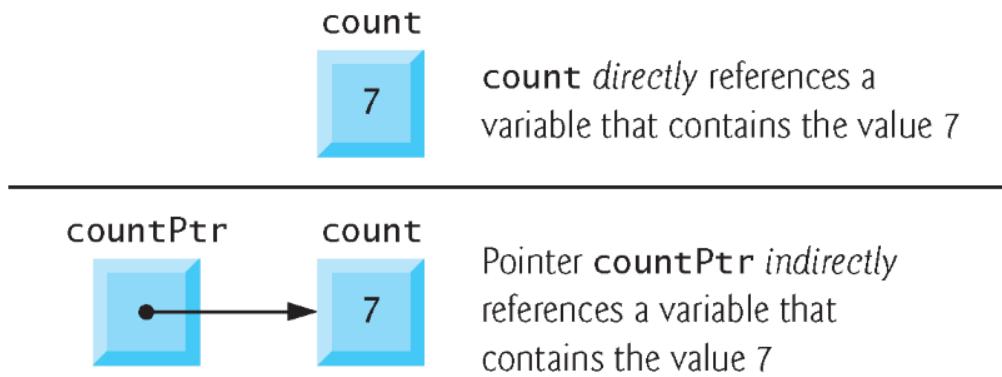


## Software Engineering Observation 8.1

*In new software development projects, you should favor array and vector objects to built-in arrays, and string objects to C strings.*

## 8.2 Pointer Variable Declarations and Initialization

- ▶ 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.

## 8.2.1 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—it indicates that the variable being declared is a pointer.
- ▶ Pointers can be declared to point to objects of *any* 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, we like to include the letters `Ptr` in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.*

## 8.2.2 Initializing Pointers

- ▶ Pointers should be initialized to `nullptr` (new in C++11) or to a memory 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.*

## 8.2.3 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

The unary operators & and \* are used to create pointer values and “dereference” pointers, respectively.

## 8.3.1 Address (&) Operator

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

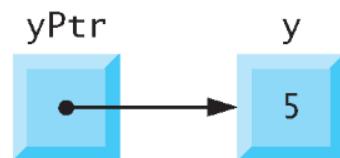
```
int y{5}; // declare variable y  
int* yPtr=nullptr; // declare pointer variable yPtr
```

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.

## 8.3.1 Address (&) Operator

- ▶ 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).



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

## 8.3.2 Indirection (\*) Operator

- ▶ The unary `*` operator—commonly referred to as the **indirection operator** or **dereferencing operator**—*returns an lvalue representing the object to which its pointer operand points.*
  - Called **dereferencing a pointer**
- ▶ A *dereferenced pointer* may also be used as an *lvalue* on the *left* side of an assignment.



## Common Programming Error 8.2

*Dereferencing an uninitialized pointer results in undefined behavior that could cause a fatal execution-time error. This could also lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results.*



## Error-Prevention Tip 8.2

*Dereferencing a null pointer results in undefined behavior and typically causes a fatal execution-time error. Ensure that a pointer is not null before dereferencing it.*

## 8.3.3 Using the Address (&) and Indirection (\*) Operators

- ▶ The program in Fig. 8.4 demonstrates the & and \* pointer operators.



## Portability Tip 8.1

*The memory addresses output by this program with cout and << are platform dependent, so you may get different results when you run the program.*

```
1 // Fig. 8.4: fig08_04.cpp
2 // Pointer operators & and *.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int a{7}; // initialize a with 7
8     int* aPtr = &a; // initialize aPtr with the address of int variable a
9
10    cout << "The address of a is " << &a
11        << "\n\nThe value of a is " << a;
12    cout << "\n\nThe value of *aPtr is " << *aPtr
13        << "\n\nThe value of *aPtr is " << *aPtr << endl;
14 }
```

```
The address of a is 002DFD80
The value of a is 7
```

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

**Fig. 8.4** | Pointer operators & and \*.

## 8.3.3 Using the Address (&) and Indirection (\*) Operators

### *Precedence and Associativity of the Operators Discussed So Far*

- ▶ Figure 8.5 lists the precedence and associativity of the operators introduced to this point.
- ▶ The address (&) and dereferencing operator (\*) are *unary operators* on the fourth level.

Operators	Associativity	Type
:: ()	left to right <i>[See caution in Fig. 2.10 regarding grouping parentheses.]</i>	primary
() [] ++ -- static_cast<type>(operand)	left to right	postfix
++ -- + - ! & *	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	stream 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 of the operators discussed so far.

## 8.4 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
  - **pass-by-reference with pointer arguments.**
- ▶ Here, we explain pass-by-reference with pointer arguments.

## 8.4 Pass-by-Reference with Pointers

- ▶ Pointers can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of copying the objects.
- ▶ You can use pointers and the indirection operator (\*) to accomplish pass-by-reference.
- ▶ When calling a function with an argument that should be modified, the *address* of the argument is passed.

## 8.4 Pass-by-Reference with Pointers (cont.)

### *An Example of Pass-By-Value*

- ▶ Figure 8.6 passes variable number by value to function `cubeByValue`, which cubes its argument and passes the new value result back to `main` using a `return` statement.
- ▶ The calling function has the opportunity to examine the function call's result before modifying any variable's values.

```
1 // Fig. 8.6: fig08_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     int number{5};
10
11     cout << "The original value of number is " << number;
12     number = cubeByValue(number); // pass number by value to cubeByValue
13     cout << "\nThe new value of number is " << number << endl;
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
18     return n * n * n; // cube local variable n and return result
19 }
```

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

**Fig. 8.6** | Pass-by-value used to cube a variable's value.

## 8.4 Pass-by-Reference with Pointers (cont.)

### *An Example of Pass-By-Reference with Pointers*

- ▶ Figure 8.7 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 uses the dereferenced pointer to cube the value to which `nPtr` points.
- ▶ This *directly* changes the value of `number` in `main`.

```
1 // Fig. 8.7: fig08_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     int number{5};
11
12     cout << "The original value of number is " << number;
13     cubeByReference(&number); // pass number address to cubeByReference
14     cout << "\nThe new value of number is " << number << endl;
15 }
16
17 // calculate cube of *nPtr; modifies variable number in main
18 void cubeByReference(int* nPtr) {
19     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
20 }
```

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

**Fig. 8.7** | Pass-by-reference with a pointer argument used to cube a variable's value.

## 8.4 Pass-by-Reference with Pointers (cont.)

### ***Insight: All Arguments Are Passed By Value***

- ▶ Passing a variable by reference with a pointer *does not actually pass anything by reference*—a pointer to that variable is *passed by value* and is *copied* into the function's corresponding pointer parameter.
- ▶ The called function can then access that variable in the caller simply by dereferencing the pointer, thus accomplishing *pass-by-reference*.

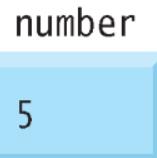
## 8.4 Pass-by-Reference with Pointers (cont.)

### *Graphical Analysis of Pass-By-Value and Pass-By-Reference*

- ▶ Figures 8.8–8.9 analyze graphically the execution of the programs in Fig. 8.6 and Fig. 8.7, respectively.
- ▶ In the diagrams, the values in blue rectangles above a given expression or variable represent the value of that expression or variable.
- ▶ Each diagram's right column shows functions `cubeByValue` (Fig. 8.6) and `cubeByReference` (Fig. 8.7) *only* when they're executing.

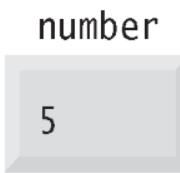
Step 1: Before main calls cubeByValue:

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

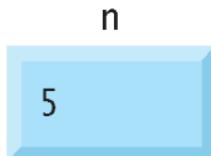


Step 2: After cubeByValue receives the call:

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



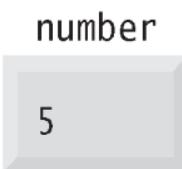
```
int cubeByValue( int n ) {  
    return n * n * n;  
}
```



**Fig. 8.8** | Pass-by-value analysis of the program of Fig. 8.6. (Part 1 of 3.)

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

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



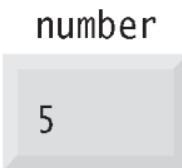
```
int cubeByValue(int n) {  
    return n * n * n;  
}
```

n

5

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

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



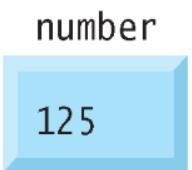
125

**Fig. 8.8** | Pass-by-value analysis of the program of Fig. 8.6. (Part 2 of 3.)

---

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

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



---

**Fig. 8.8** | Pass-by-value analysis of the program of Fig. 8.6. (Part 3 of 3.)

Step 1: Before main calls cubeByReference:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

5

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

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```

number

5

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

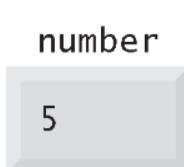
*call establishes this pointer*

nPtr

**Fig. 8.9** | Pass-by-reference analysis of the program of Fig. 8.7. (Part 1 of 3.)

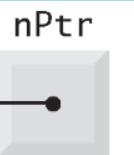
Step 3: Before `*nPtr` is assigned the result of the calculation  $5 * 5 * 5$ :

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```



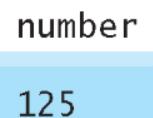
```
void cubeByReference(int* nPtr) {  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}
```

125

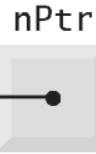


Step 4: After `*nPtr` is assigned 125 and before program control returns to `main`:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```



```
void cubeByReference(int* nPtr) {  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}  
called function modifies caller's  
variable
```

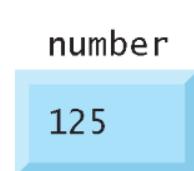


**Fig. 8.9** | Pass-by-reference analysis of the program of Fig. 8.7. (Part 2 of 3.)

---

Step 5: After `cubeByReference` returns to `main`:

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```



---

**Fig. 8.9** | Pass-by-reference analysis of the program of Fig. 8.7. (Part 3 of 3.)

## 8.5 Built-In Arrays

- Here we present *built-in arrays*, which are also *fixed-size* data structures.

## 8.5.1 Declaring and Accessing a Built-In Array

- ▶ To specify the type of the elements and the number of elements required by a built-in array, use a declaration of the form:  
*type arrayName[arraySize];*
- ▶ The compiler reserves the appropriate amount of memory.
- ▶ *arraySize* must be an integer constant greater than zero.
- ▶ To reserve 12 elements for built-in array of **ints** named **c**, use the declaration
  - `int c[12];`
- ▶ As with array objects, you use the subscript (`[]`) operator to access the individual elements of a built-in array.

## 8.5.2 Initializing Built-In Arrays

- ▶ You can initialize the elements of a built-in array using an initializer list as in
  - `int n[5]{50, 20, 30, 10, 40};`
  - creates a built-in array of five `ints` and initializes them to the values in the initializer list.
- ▶ If you provide fewer initializers
  - the number of elements, the remaining elements are value initialized—fundamental numeric types are set to 0, `bools` are set to false, pointers are set to `nullptr` and class objects are initialized by their default constructors.
- ▶ If you provide too many initializers a compilation error occurs.

## 8.5.2 Initializing Built-In Arrays

- ▶ If a built-in array's size is *omitted* from a declaration with an initializer list, the compiler sizes the built-in array to the number of elements in the initializer list.
- ▶ For example,

```
int n[] {50, 20, 30, 10, 40};
```
- ▶ creates a five-element array.



### Error-Prevention Tip 8.3

*Always specify a built-in array's size, even when providing an initializer list. This enables the compiler to generate an error message if there are more initializers than array elements.*

## 8.5.3 Passing Built-In Arrays to Functions

- ▶ *The value of a built-in array's name is implicitly convertible to the address of the built-in array's first element.*
  - So `arrayName` is implicitly convertible to `&arrayName[0]`.
- ▶ You don't need to take the address (`&`) of a built-in array to pass it to a function—you simply pass the built-in array's name.
- ▶ For built-in arrays, the called function can modify *all* the elements of a built-in array in the caller—unless the function precedes the corresponding built-in array parameter with `const` to indicate that the elements should *not* be modified.



## Software Engineering Observation 8.2

*Applying the `const` type qualifier to a built-in array parameter in a function definition to prevent the original built-in array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify a built-in array unless it's absolutely necessary.*

## 8.5.4 Declaring Built-In Array Parameters

- ▶ You can declare a built-in array parameter in a function header, as follows:

```
int sumElements(const int values[], const size_t  
    numberOfElements)
```

- ▶ which indicates that the function's first argument should be a one-dimensional built-in array of `ints` that should not be modified by the function.
- ▶ The preceding header can also be written as:

```
int sumElements(const int* values, const size_t  
    numberOfElements)
```

## 8.5.4 Declaring Built-In Array Parameters

- ▶ *The compiler does not differentiate between a function that receives a pointer and a function that receives a built-in array.*
  - The function must “know” when it’s receiving a built-in array or simply a single variable that’s being passed by reference.
- ▶ When the compiler encounters a function parameter for a one-dimensional built-in array of the form `const int values[]`, the compiler converts the parameter to the pointer notation `const int* values`.
  - These forms of declaring a one-dimensional built-in array parameter are interchangeable.



## Good Programming Practice 8.2

*When declaring a built-in array parameter, for clarity  
use the [] notation rather than pointer notation.*

## 8.5.5 C++11: Standard Library Functions `begin` and `end`

- ▶ In Section 7.7, we showed how to sort an array object with the C++ Standard Library function `sort`.
- ▶ We sorted an **array of strings** called `colors` as follows:

```
// sort contents of colors  
sort(colors.begin(), colors.end());
```

- ▶ The **array class's** `begin` and `end` functions specified that the entire array should be sorted.

## 8.5.5 C++11: Standard Library Functions `begin` and `end`

- ▶ Function `sort` (and many other C++ Standard Library functions) can also be applied to built-in arrays.
- ▶ For example, to sort the built-in array `n` shown earlier in this section, you can write:

```
// sort contents of built-in array n  
sort(begin(n), end(n));
```

- ▶ C++11's new `begin` and `end` functions (from header `<iterator>`) each receive a built-in array as an argument and return a pointer that can be used to represent ranges of elements to process in C++ Standard Library functions like `sort`.

## 8.5.6 Built-In Array Limitations

- ▶ Built-in arrays have several limitations:
  - They *cannot be compared* using the relational and equality operators—you must use a loop to compare two built-in arrays element by element.
  - They *cannot be assigned* to one another.
  - They *don't know their own size*—a function that processes a built-in array typically receives *both* the built-in array's *name* and its *size* as arguments.
  - They *don't provide automatic bounds checking*—you must ensure that array-access expressions use subscripts that are within the built-in array's bounds.

## 8.5.7 Built-In Arrays Sometimes Are Required

- ▶ There are cases in which built-in arrays *must* be used, such as processing a program's **command-line arguments**.
- ▶ You supply command-line arguments to a program by placing them after the program's name when executing it from the command line. Such arguments typically pass options to a program.

## 8.5.7 Built-In Arrays Sometimes Are Required

- ▶ On a Windows computer, the command

```
dir /p
```

- ▶ uses the /p argument to list the contents of the current directory, pausing after each screen of information.
- ▶ On Linux or OS X, the following command uses the -la argument to list the contents of the current directory with details about each file and directory:

```
ls -la
```

## 8.6 Using `const` with Pointers

- ▶ Many possibilities exist for using (or not using) `const` with function parameters.
- ▶ *Principle of least privilege*
  - Always give a function *enough* access to the data in its parameters to accomplish its specified task, *but no more*.



## Software Engineering Observation 8.3

*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`.*



## Error-Prevention Tip 8.4

*Before using a function, check its function prototype to determine the parameters that it can and cannot modify.*

## 8.6 Using `const` with Pointers (cont.)

- ▶ There are four ways to pass a pointer to a function
  - a nonconstant pointer to nonconstant data
  - a nonconstant pointer to constant data (Fig. 8.10)
  - a constant pointer to nonconstant data (Fig. 8.11)
  - a constant pointer to constant data (Fig. 8.12)
- ▶ Each combination provides a different level of access privilege.

## 8.6.1 Nonconstant Pointer to Nonconstant Data

- ▶ The highest access is granted by a **nonconstant pointer to nonconstant data**
  - The *data can be modified* through the dereferenced pointer
  - The pointer can be modified to point to other data.
- ▶ Such a pointer's declaration (e.g., `int* countPtr`) does not include `const`.

## 8.6.2 Nonconstant Pointer to Constant Data

- ▶ **Nonconstant pointer to constant data**
  - A pointer that can be modified to point to any item of the appropriate type
  - The data to which it points *cannot* be modified through that pointer
- ▶ **Sample declaration:**

```
const int* countPtr;
```

  - Read from *right to left* as “`countPtr` is a pointer to an integer constant” or more precisely, “`countPtr` is a non-constant pointer to an integer constant.”
- ▶ Figure 8.10 demonstrates GNU C++’s compilation error message produced when attempting to compile a function that receives a *nonconstant pointer* to *constant data*, then tries to use that pointer to modify the data.

---

```
1 // Fig. 8.10: fig08_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     int y{0};
9
10    f(&y); // f will attempt an illegal modification
11 }
12
13 // constant variable cannot be modified through xPtr
14 void f(const int* xPtr) {
15     *xPtr = 100; // error: cannot modify a const object
16 }
```

*GNU C++ compiler error message:*

```
fig08_10.cpp: In function ‘void f(const int*)’:
fig08_10.cpp:17:12: error: assignment of read-only location ‘* xPtr’
```

**Fig. 8.10** | Attempting to modify data through a nonconstant pointer to `const` data.



## 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 and avoid the copy overhead of pass-by-value.*



## Software Engineering Observation 8.4

*Passing large objects using pointers to constant data, or references to constant data, offers the security of pass-by-value.*



## Software Engineering Observation 8.5

*Use pass-by-value to pass fundamental-type arguments (e.g., ints, doubles, etc.) to a function unless the caller explicitly requires that the called function be able to directly modify the value in the caller. This is another example of the principle of least privilege.*

## 8.6.3 Constant Pointer to Nonconstant Data

- ▶ Constant pointer to nonconstant data
  - always points to the same memory location
  - the data at that location can be modified through the pointer
- ▶ Pointers that are declared `const` *must be initialized when they're declared.*
- ▶ If the pointer is a function parameter, it's *initialized with a pointer that's passed to the function.*
- ▶ Fig. 8.11 attempts to modify a constant pointer.

```
1 // Fig. 8.11: fig08_11.cpp
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11    *ptr = 7; // allowed: *ptr is not const
12    ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

*Microsoft Visual C++ compiler error message:*

```
'ptr': you cannot assign to a variable that is const
```

**Fig. 8.11** | Attempting to modify a constant pointer to nonconstant data.

## 8.6.4 Constant Pointer to Constant Data

- ▶ The *minimum* access privilege is granted by a **constant pointer to constant data**.
  - Such a pointer *always* points to the *same* memory location
  - The data at that location *cannot* be modified via the pointer.
- ▶ This is how a built-in array should be passed to a function that *only reads* from the built-in array, using array subscript notation, and *does not modify* the built-in array.
- ▶ **const int\* const**
  - Read from *right to left* as “ptr is a *constant pointer to an integer constant*.”
- ▶ Fig. 8.12 shows the Xcode LLVM compiler’s error messages that are generated when an attempt is made to modify the data to which ptr points and when an attempt is made to modify the address stored in the pointer variable—these show up on the lines of code with the errors in the Xcode text editor.

```
1 // Fig. 8.12: fig08_12.cpp
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int x{5}, y;
8
9     // ptr is a constant pointer to a constant integer.
10    // ptr always points to the same location; the integer
11    // at that location cannot be modified.
12    const int* const ptr{&x};
13
14    cout << *ptr << endl;
15
16    *ptr = 7; // error: *ptr is const; cannot assign new value
17    ptr = &y; // error: ptr is const; cannot assign new address
18 }
```

Xcode LLVM compiler error message:

```
Read-only variable is not assignable
Read-only variable is not assignable
```

**Fig. 8.12** | Attempting to modify a constant pointer to constant data.

## 8.7 `sizeof` Operator

- ▶ The unary operator `sizeof` determines the size in bytes of a built-in array or of any other data type, variable or constant *during program compilation*.
- ▶ When applied to a built-in array's name, as in Fig. 8.13, the `sizeof` operator returns the *total number of bytes in the built-in array* as a value of type `size_t`.
- ▶ When applied to a *pointer parameter* in a function that *receives a built-in array as an argument*, the `sizeof` operator returns the size of the *pointer* in bytes—not the built-in array's size.



## Common Programming Error 8.3

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

---

```
1 // Fig. 8.13: fig08_13.cpp
2 // Sizeof operator when used on a built-in array's name
3 // returns the number of bytes in the built-in array.
4 #include <iostream>
5 using namespace std;
6
7 size_t getSize(double*); // prototype
8
9 int main() {
10     double numbers[20]; // 20 doubles; occupies 160 bytes on our system
11
12     cout << "The number of bytes in the array is " << sizeof(numbers);
13
14     cout << "\nThe number of bytes returned by getSize is "
15         << getSize(numbers) << endl;
16 }
17
18 // return size of ptr
19 size_t getSize(double* ptr) {
20     return sizeof(ptr);
21 }
```

---

**Fig. 8.13** | `sizeof` operator when applied to a built-in array's name returns the number of bytes in the built-in array. (Part I of 2.)

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

**Fig. 8.13** | `sizeof` operator when applied to a built-in array's name returns the number of bytes in the built-in array. (Part 2 of 2.)

## 8.7 sizeof Operator (cont.)

- ▶ To determine the number of elements in the built-in array `numbers`, use the following expression (which is evaluated at *compile time*) :
  - `sizeof numbers / sizeof(numbers[0])`
- ▶ The expression divides the number of bytes in `numbers` by the number of bytes in the built-in array's zeroth element.

## 8.7 sizeof Operator (cont.)

- ▶ Figure 8.14 uses `sizeof` to calculate the number of bytes used to store many of the standard data types.
- ▶ The output was produced using the default settings in Xcode 7.2 on Mac OS X.
- ▶ Type sizes are platform dependent.

---

```
1 // Fig. 8.14: fig08_14.cpp
2 // sizeof operator used to determine standard data type sizes.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     char c; // variable of type char
8     short s; // variable of type short
9     int i; // variable of type int
10    long l; // variable of type long
11    long long ll; // variable of type long 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]; // built-in array of int
16    int* ptr{array}; // variable of type int*
17
18    cout << "sizeof c = " << sizeof c
```

---

**Fig. 8.14** | sizeof operator used to determine standard data type sizes. (Part 1 of 3.)

---

```
19    << "\tsizeof(char) = " << sizeof(char)
20    << "\nsizeof s = " << sizeof s
21    << "\tsizeof(short) = " << sizeof(short)
22    << "\nsizeof i = " << sizeof i
23    << "\tsizeof(int) = " << sizeof(int)
24    << "\nsizeof l = " << sizeof l
25    << "\tsizeof(long) = " << sizeof(long)
26    << "\nsizeof ll = " << sizeof ll
27    << "\tsizeof(long long) = " << sizeof(long long)
28    << "\nsizeof f = " << sizeof f
29    << "\tsizeof(float) = " << sizeof(float)
30    << "\nsizeof d = " << sizeof d
31    << "\tsizeof(double) = " << sizeof(double)
32    << "\nsizeof ld = " << sizeof ld
33    << "\tsizeof(long double) = " << sizeof(long double)
34    << "\nsizeof array = " << sizeof array
35    << "\nsizeof ptr = " << sizeof ptr << endl;
36 }
```

---

**Fig. 8.14** | sizeof operator used to determine standard data type sizes. (Part 2 of 3.)

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

**Fig. 8.14** | sizeof operator used to determine standard data type sizes. (Part 3 of 3.)



## Portability Tip 8.2

*The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, always use `sizeof` to determine the number of bytes used to store the data types.*

## 8.7 **sizeof** Operator (cont.)

- ▶ Operator **sizeof** can be applied to any expression or type name.
- ▶ When **sizeof** is applied to a variable name (which is not a built-in array's name) or other expression, the number of bytes used to store the specific type of the expression is returned.
- ▶ The parentheses used with **sizeof** are required *only* if a type name is supplied as its operand.

## 8.8 Pointer Expressions and Pointer Arithmetic

- ▶ Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- ▶ C++ enables **pointer arithmetic**—a few 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—this particular operation is appropriate only for two pointers that point to elements of the same built-in array



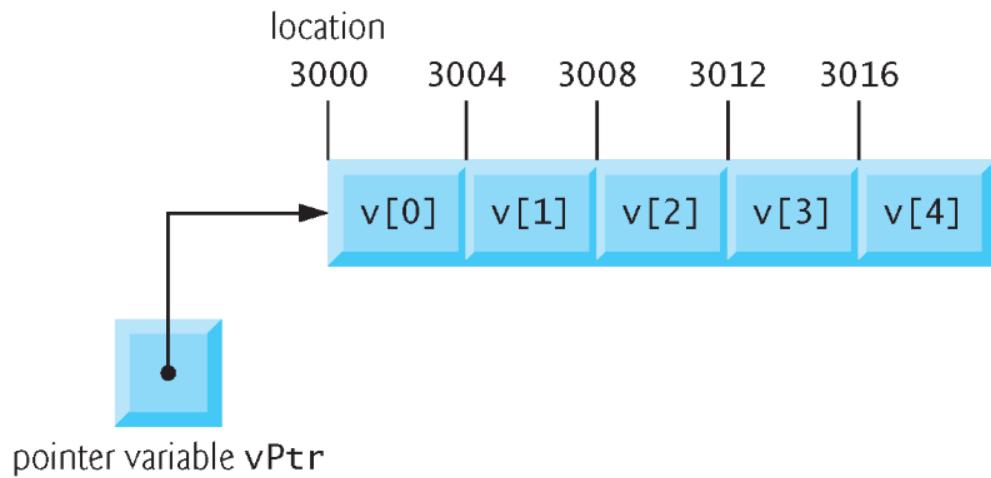
## Portability Tip 8.3

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

## 8.8 Pointer Expressions and Pointer Arithmetic

- ▶ Assume that `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 8.15 diagrams this situation for a machine with four-byte integers. Variable `vPtr` can be initialized to point to `v` with either of the following statements:

```
int* vPtr{v};  
int* vPtr{&v[0]};
```



---

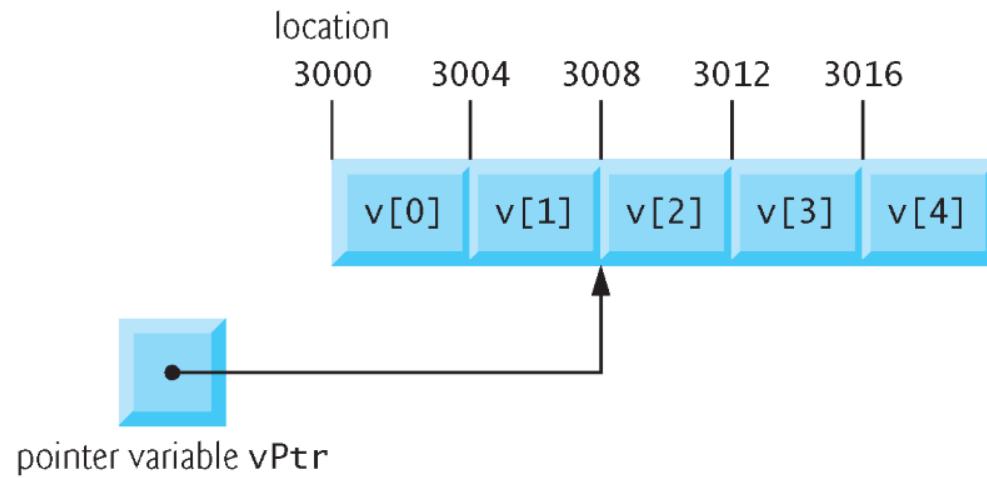
**Fig. 8.15** | Built-in array `v` and a pointer variable `int* vPtr` that points to `v`.

## 8.8.1 Adding Integers to and Subtracting Integers from Pointers

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

## 8.8.1 Adding Integers to and Subtracting Integers from Pointers

- ▶ `vPtr += 2;`
  - would produce 3008 (from the calculation  $3000 + 2 * 4$ ), assuming that an int is stored in four bytes of memory.
- ▶ In the built-in array `v`, `vPtr` would now point to `v[2]` (Fig. 8.16).
- ▶ If an integer is stored in eight bytes of memory, then the preceding calculation would result in memory location 3016 ( $3000 + 2 * 8$ ).



---

**Fig. 8.16** | Pointer  $vPtr$  after pointer arithmetic.



## Error-Prevention Tip 8.5

*There's no bounds checking on pointer arithmetic. You must ensure that every pointer arithmetic operation that adds an integer to or subtracts an integer from a pointer results in a pointer that references an element within the built-in array's bounds.*

## 8.8.3 Subtracting Pointers

- ▶ Pointer variables pointing to the *same* built-in 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 built-in array elements from `vPtr` to `v2Ptr`—in this case, 2.
- ▶ *Pointer arithmetic is meaningful only on a pointer that points to a built-in array.*



## Common Programming Error 8.4

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

## 8.8.3 Pointer Assignment

- ▶ A pointer can be assigned to another pointer if both pointers are of the *same* type.
- ▶ Otherwise, a cast operator (normally a `reinterpret_cast`; discussed in Section 14.7) must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment.
  - Exception to this rule is the **pointer to void** (i.e., `void*`).
- ▶ *Any pointer to a fundamental type or class type can be assigned to a pointer of type `void*` without casting.*



## Common Programming Error 8.5

*Assigning a pointer of one type to a pointer of another (other than void\*) without using a cast is a compilation error.*

## 8.8.4 Cannot Dereference a `void*`

- ▶ A `void*` pointer *cannot* be dereferenced.
- ▶ The compiler “knows” that an `int*` points to four bytes of memory on a machine with four-byte integers—dereferencing an `int*` creates an *lvalue* that is an alias for the `int`’s four bytes in memory.
- ▶ A `void*` simply contains a memory address for an unknown data type.
  - You cannot dereference a `void*` because the compiler does not know the type of the data to which the pointer refers and thus not the number of bytes.



## Common Programming Error 8.6

*The allowed operations on void\* pointers are: comparing void\* pointers with other pointers, casting void\* pointers to other pointer types and assigning addresses to void\* pointers. All other operations on void\* pointers are compilation errors.*

## 8.8.5 Comparing Pointers

- ▶ Pointers can be compared using equality and relational operators.
  - Comparisons using relational operators are meaningless unless the pointers point to elements of the *same* built-in array.
  - Pointer comparisons compare the *addresses* stored in the pointers.
- ▶ A common use of pointer comparison is determining whether a pointer has the value `nullptr`, `0` or `NULL` (i.e., the pointer does not point to anything).

## 8.9 Relationship Between Pointers and Built-In Arrays

- ▶ Pointers can be used to do any operation involving array subscripting.
- ▶ Assume the following declarations:

```
// create 5-element int array b; b is a const pointer
int b[5];
// create int pointer bPtr, which isn't a const pointer
int* bPtr;
```

## 8.9 Relationship Between Pointers and Built-In Arrays

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

```
// assign address of built-in array b to bPtr  
bPtr = b;
```

- ▶ This is equivalent to assigning the address of the first element as follows:

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

## 8.9.1 Pointer/Offset Notation

- ▶ Built-in array element `b[3]` can alternatively be referenced with the pointer expression
  - `*(bPtr + 3)`
- ▶ The `3` 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 `+`.

## 8.9 Relationship Between Pointers and Built-In Arrays (cont.)

- ▶ Just as the built-in array element can be referenced with a pointer expression, the *address*
  - `&b[3]`
- ▶ can be written with the pointer expression
  - `bPtr + 3`

## 8.9.2 Pointer/Offset Notation with the Built-In Array's Name as the Pointer

- ▶ The built-in array name can be treated as a pointer and used in pointer arithmetic.
- ▶ For example, the expression
  - $*(b + 3)$
- ▶ also refers to the element  $b[3]$ .
- ▶ In general, all subscripted built-in array expressions can be written with a pointer and an offset.

## 8.9.3 Pointer/Subscript Notation

- ▶ Pointers can be subscripted exactly as built-in arrays can.
- ▶ For example, the expression
  - `bPtr[1]`
- ▶ refers to `b[1]`; this expression uses **pointer/subscript notation**.



## Good Programming Practice 8.3

*For clarity, use built-in array notation instead of pointer notation when manipulating built-in arrays.*

## 8.9.4 Demonstrating the Relationship Between Pointers and Built-In Arrays

- ▶ Figure 8.17 uses the four notations we just discussed
  - array subscript notation
  - pointer/offset notation with the built-in array's name as a pointer
  - pointer subscript notation
  - pointer/offset notation with a pointer
- ▶ to accomplish the same task, namely displaying the four elements of the built-in array of `ints` named `b`.

```
1 // Fig. 8.17: fig08_17.cpp
2 // Using subscripting and pointer notations with built-in arrays.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int b[]{10, 20, 30, 40}; // create 4-element built-in array b
8     int* bPtr{b}; // set bPtr to point to built-in array b
9
10    // output built-in array b using array subscript notation
11    cout << "Array b displayed with:\n\nArray subscript notation\n";
12
13    for (size_t i{0}; i < 4; ++i) {
14        cout << "b[" << i << "] = " << b[i] << '\n';
15    }
16
17    // output built-in array b using array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19        << "the pointer is the array name\n";
20
21    for (size_t offset1{0}; offset1 < 4; ++offset1) {
22        cout << "*(" << b << offset1 << ") = " << *(b + offset1) << '\n';
23    }
```

**Fig. 8.17** | Using subscripting and pointer notations with built-in arrays. (Part 1 of 3.)

---

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

---

**Fig. 8.17** | Using subscripting and pointer notations with built-in arrays. (Part 2 of 3.)

Array b displayed 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
```

**Fig. 8.17** | Using subscripting and pointer notations with built-in arrays. (Part 3 of 3.)

## 8.10 Pointer-Based Strings

- ▶ This section introduces C-style, pointer-based strings, which we'll simply call **C strings**.
- ▶ *C++'s string class is preferred*
  - *it eliminates many of the security problems that can be caused by manipulating C strings.*
- ▶ We cover C strings for a deeper understanding of arrays, and because there are some cases in which C string processing is required.
- ▶ Also, if you work with legacy C and C++ programs, you're likely to encounter pointer-based strings.

## 8.10 Pointer-Based Strings (cont.)

### ***Characters and Character Constants***

- ▶ Characters are the fundamental building blocks of C++ source programs.
- ▶ **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.

## 8.10 Pointer-Based Strings (cont.)

### *Strings*

- ▶ 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

## 8.10 Pointer-Based Strings (cont.)

### *Pointer-Based Strings*

- ▶ A pointer-based string is a built-in array of characters ending with a **null character** ('`\0`').
- ▶ A string is accessed via a pointer to its first character.
- ▶ The `sizeof` a string literal is the length of the string including the terminating null character.

## 8.10 Pointer-Based Strings (cont.)

### *String Literals as Initializers*

- ▶ A string literal may be used as an initializer in the declaration of either a built-in array of chars or a variable of type `const char*`.
- ▶ String literals exist for the duration of the program and may be *shared* if the same string literal is referenced from multiple locations in a program.

## 8.10 Pointer-Based Strings (cont.)

### ***Character Constants as Initializers***

- ▶ When declaring a built-in array of chars to contain a string, the built-in array must be large enough to store the string *and* its terminating null character.



## Common Programming Error 8.7

*Not allocating sufficient space in a built-in array of chars to store the null character that terminates a string is a logic error.*



## Common Programming Error 8.8

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



## Error-Prevention Tip 8.6

*When storing a string of characters in a built-in array of chars, be sure that the built-in array is large enough to hold the largest string that will be stored. C++ allows strings of any length. If a string is longer than the built-in array of chars in which it's to be stored, characters beyond the end of the built-in array will overwrite data in memory following the built-in array, leading to logic errors and potential security breaches.*

## 8.10 Pointer-Based Strings (cont.)

### *Accessing Characters in a C String*

- ▶ Because a C string is a built-in array of characters, we can access individual characters in a string directly with array subscript notation.

## 8.10 Pointer-Based Strings (cont.)

### *Reading Strings into Built-In Arrays of char with cin*

- ▶ A string can be read into a built-in array of chars using stream extraction with `cin`.
- ▶ The `setw` stream manipulator can be used to *ensure* that the string read into word *does not exceed the size of the built-in array*.
  - Applies *only* to the next value being input.

## 8.10 Pointer-Based Strings (cont.)

### *Reading Lines of Text into Built-In Arrays of char with cin.getline*

- ▶ In some cases, it's desirable to input an *entire line of text* into a built-in array of chars.
- ▶ For this purpose, the `cin` object provides the member function `getline`, which takes three arguments—a *built-in array of chars* in which the line of text will be stored, a *length* and a *delimiter character*.
- ▶ The function stops reading characters when the delimiter character '`\n`' is encountered, when the *end-of-file indicator* is entered or when the number of characters read so far is one less than the length specified in the second argument.
- ▶ The third argument to `cin.getline` has '`\n`' as a default value.

## 8.10 Pointer-Based Strings (cont.)

### *Displaying C Strings*

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

## 8.10 Pointer-Based Strings (cont.)

### *Displaying C Strings*

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

## 8.11 Note About Smart Pointers

- ▶ Later in the book, we introduce dynamic memory management with pointers, which allows you at execution time to create and destroy objects as needed.
- ▶ Improperly managing this process is a source of subtle errors.
- ▶ We'll discuss "smart pointers," which help you avoid dynamic memory management errors by providing additional functionality beyond that of built-in pointers.