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. (Part 2 of 2.)

Performance Tip 8.1

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

Software Engineering Observation 8.3

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

Software Engineering Observation 8.4

Use pass-by-value to pass fundamental-type arguments (e.g., `int`s, `double`s, 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

- A constant pointer to nonconstant data is a pointer that always points to the same memory location, and 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.*
- The program of 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  {
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:

```
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, and 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.
- The program of Fig. 8.12 declares pointer variable `ptr` to be of type `const int * const` (line 13).
- This declaration is read from *right to left* as "`ptr` is a *constant pointer to an integer constant*."
- The figure 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  {
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
```

Fig. 8.12 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

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. (Part 2 of 2.)

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*

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 {
11    double numbers[ 20 ]; // 20 doubles; occupies 160 bytes on our system
12
13    cout << "The number of bytes in the array is " << sizeof( numbers );
14
15    cout << "\nThe number of bytes returned by getSize is "
16       << getSize( numbers ) << 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
```

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 1 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 Visual C++ 2012 on a Windows 7 computer.
  - Type sizes are platform dependent.