8.8 Pointer Expressions and Pointer Arithmetic (cont.)

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.*

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- A `void *` pointer *cannot* be dereferenced.
  - The compiler must know the data type to determine the number of bytes to dereference for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined.

Common Programming Error 8.5

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

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 Pointer Expressions and Pointer Arithmetic (cont.)

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

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

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

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 8.2

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

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

Demonstrating the Relationship Between Pointers and Built-In Arrays

- Figure 8.17 uses the four notations discussed in this section for referring to built-in array elements—*array subscript notation, pointer/offset notation with the built-in array's name as a pointer, pointer subscript notation and pointer/offset notation with a pointer*—to accomplish the same task, namely displaying the four elements of the built-in array of `int`s 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  {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element built-in array b
9     int *bPtr = b; // set bPtr to point to built-in array b
10
11    // output built-in array b using array subscript notation
12    cout << "Array b displayed with:\n\nArray subscript notation\n";
13
14    for ( size_t i = 0; i < 4; ++i )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
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 4.)

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

Fig. 8.17 | Using subscripting and pointer notations with built-in arrays. (Part 2 of 4.)