```
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  {
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    long ll; // variable of type long long
13    float f; // variable of type float
14    double d; // variable of type double
15    long double ld; // variable of type long double
16    int array[ 20 ]; // built-in array of int
17    int *ptr = array; // variable of type int *
18
```

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

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

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 = 4     sizeof(long) = 4
sizeof ll = 8    sizeof(long long) = 8
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. 8.14 | `sizeof` operator used to determine standard data type sizes. (Part 3 of 3.)

Portability Tip 8.1

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 8.2

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.

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:

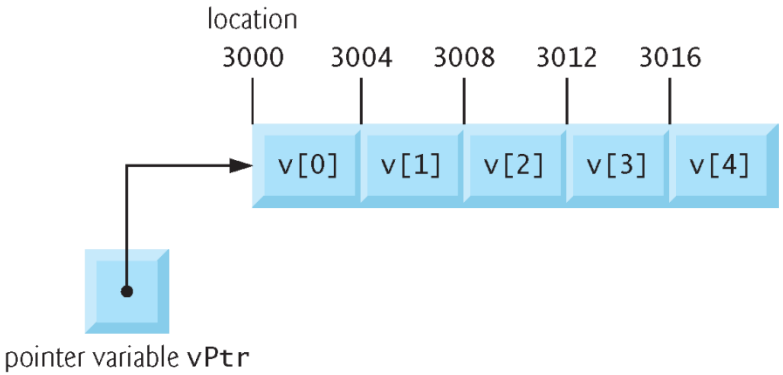

Fig. 8.15 | Built-in array `v` and a pointer variable `int *vPtr` that points to `v`.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

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 object to which the pointer refers*.
  - The number of bytes depends on the object's data

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- For example, the statement

```
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`

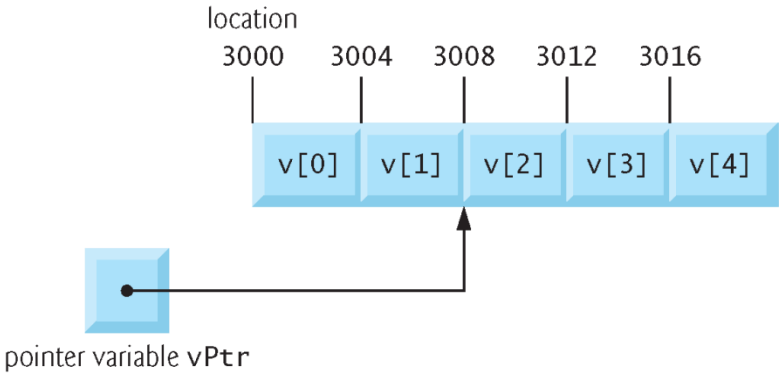

Fig. 8.16 | Pointer `vPtr` after pointer arithmetic.

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 Pointer Expressions and Pointer Arithmetic (cont.)

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.

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.