Software Engineering Observation 8.1

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 Built-In Arrays (cont.)

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 `int`s 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 Built-In Arrays (cont.)

- *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.

8.5 Built-In Arrays (cont.)

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 `string`s 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 Built-In Arrays (cont.)

- 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 Built-In Arrays (cont.)

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.
- Objects of class templates `array` and `vector` are safer, more robust and provide more capabilities than built-in arrays.

8.5 Built-In Arrays (cont.)

Sometimes Built-In Arrays 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 Built-In Arrays (cont.)

- 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 8.2

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, and 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

- A nonconstant pointer to constant data
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points *cannot* be modified through that pointer.
- Might be used to *receive* a built-in array argument to a function that should be allowed to read the elements, but *not* modify them.
- Any attempt to modify the data in the function results in a compilation error.
- 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.

```cpp
// Fig. 8.10: fig08_10.cpp
// Attempting to modify data through a
// nonconstant pointer to constant data.

void f( const int * ); // prototype

int main()
{
   int y = 0;

   f( &y ); // f will attempt an illegal modification
} // end main

// constant variable cannot be modified through xPtr
void f( const int *xPtr )
{
   *xPtr = 100; // error: cannot modify a const object
} // end function f
```

Fig. 8.10 | Attempting to modify data through a nonconstant pointer to `const` data. (Part 1 of 2.)