Common Programming Error 10.5

Not providing a copy constructor and overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a potential logic error.

10.10 Case Study: `Array` Class (cont.)

C++11: Deleting Unwanted Member Functions from Your Class

- Prior to C++11, you could prevent class objects from being *copied* or *assigned* by declaring as `private` the class's copy constructor and overloaded assignment operator.
- As of C++11, you can simply *delete* these functions from your class.
- To do so in class `Array`, replace the prototypes in lines 15 and 19 of Fig. 10.10 with:

```
Array( const Array & ) = delete;
const Array &operator=( const Array & ) = delete;
```

- Though you can delete any member function, it's most commonly used with member functions that the compiler can auto-generate—the default constructor, copy constructor, assignment operator, and in C++ 11, the move constructor and move assignment operator.

10.10 Case Study: `Array` Class (cont.)

Overloaded Equality and Inequality Operators

- Line 20 of Fig. 10.10 declares the overloaded equality operator (==) for the class.
- When the compiler sees the expression `integers1 == integers2` in line 55 of Fig. 10.9, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Member function `operator==` (defined in Fig. 10.11, lines 66–76) immediately returns `false` if the `size` members of the `Array`s are not equal.
- Otherwise, `operator==` compares each pair of elements.
- If they're all equal, the function returns `true`.
- The first pair of elements to differ causes the function to return `false` immediately.

10.10 Case Study: `Array` Class (cont.)

- Lines 23–26 Fig. 10.9 define the overloaded inequality operator (`!=`) for the class.
- Member function `operator!=` uses the overloaded `operator==` function to deter-mine whether one `Array` is equal to another, then returns the opposite of that result.
- Writing `operator!=` in this manner enables you to reuse `operator==`, which *reduces the amount of code that must be written in the class*.
- Also, the full function definition for `operator!=` is in the `Array` header.
  - Allows the compiler to inline the definition.

10.10 Case Study: `Array` Class (cont.)

Overloaded Subscript Operators

- Lines 29 and 32 of Fig. 10.10 declare two overloaded subscript operators (defined in Fig. 10.11 in lines 80–87 and 91–98).
- When the compiler sees the expression `integers1[5]` (Fig. 10.9, line 59), it invokes the appropriate overloaded `operator[]` member function by generating the call
  - `integers1.operator[]( 5 )`
- The compiler creates a call to the `const` version of `operator[]` (Fig. 10.11, lines 91–98) when the subscript operator is used on a `const` `Array` object.

10.10 Case Study: `Array` Class (cont.)

- Each definition of `operator[]` determines whether the subscript it receives as an argument is in range.
- If it isn't, each function prints an error message and terminates the program with a call to function `exit`.
- If the subscript is in range, the non-`const` version of `operator[]` returns the appropriate `Array` element as a reference so that it may be used as a modifiable *lvalue.*
- If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the `Array`.

10.10 Case Study: `Array` Class (cont.)

C++11: Managing Dynamically Allocated Memory with `unique_ptr`

- In this case study, class `Array`'s destructor used `delete []` to return the dynamically allocated built-in array to the free store.
- As you recall, C++11 enables you to use `unique_ptr` to ensure that this dynamically allocated memory is deleted when the `Array` object goes out of scope.

10.10 Case Study: `Array` Class (cont.)

C++11: Passing a List Initializer to a Constructor

- In Fig. 7.4, we showed how to initialize an array object with a comma-separated list of initializers in braces, as in

```
array< int, 5 > n = { 32, 27, 64, 18, 95 };
```

- C++11 now allows any object to be initialized with a list initializer and that the preceding statement can also be written without the =, as in

```
array< int, 5 > n{ 32, 27, 64, 18, 95 };
```

10.10 Case Study: `Array` Class (cont.)

- C++11 also allows you to use list initializers when you declare objects of your own classes.
- For example, you can now provide an Array constructor that would enabled the following declarations:

```
Array integers = { 1, 2, 3, 4, 5 };
```

- or

```
Array integers{ 1, 2, 3, 4, 5 };
```

- each of which creates an `Array` object with five elements containing the integers from 1 to 5.

10.10 Case Study: `Array` Class (cont.)

- To support list initialization, you can define a constructor that receives an *object* of the class template `initializer_list`. For class `Array`, you'd include the `<initializer_list>` header.
- Then, you'd define a constructor with the first line:

```
Array::Array( initializer_list< int > list )
```

- You can determine the number of elements in the list parameter by calling its size member function.
- To obtain each initializer and copy it into the `Array` object's dynamically allocated built-in array, you can use a range-based for as follows:

```
size_t i = 0;
for ( int item : list )
   ptr[ i++ ] = item;
```

10.11 Operators as Member vs. Non-Member Functions

- Whether an operator function is implemented as a *member function* or as a non-member function, the operator is still used the same way in expressions.
- When an operator function is implemented as a *member function*, the *leftmost* (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand *must* be an object of a different class or a fundamental type, this operator function *must* be implemented as a non-member function (as we did in Section 10.5 when overloading `<<` and `>>` as the stream insertion and extraction operators, respectively).
- A non-member operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.
- Operator member functions of a specific class are called only when the left operand of a binary operator is specifically an object of that class, or when the *single operand of a unary operator* is an object of that class.

10.11 Operators as Member vs. Non-Member Functions (cont.)

- You might choose a non-member function to overload an operator to enable the operator to be *commutative*.
- The `operator+` function that deals with the `HugeInt` on the left, can still be a *member function*.
- The *non-member function* simply swaps its arguments and calls the *member function*.

10.12 Converting Between Types

- Sometimes all the operations "stay within a type." For example, adding an `int` to an `int` produces an `int`.
- It's often necessary, however, to convert data of one type to data of another type.
- The compiler knows how to perform certain conversions among fundamental types.
- You can use *cast operators* to *force* conversions among fundamental types.
- The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this.

10.12 Converting Between Types (cont.)

- Such conversions can be performed with conversion constructors—constructors that can be called with a single argument (we'll refer to these as *single-argument constructors*).
- Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

10.12 Converting Between Types (cont.)

Conversion Operators

- A conversion operator (also called a *cast operator*) can be used to convert an object of one class to another type.
- Such a conversion operator must be a *non-`static` member function.*
- The function prototype
  - `MyClass::operator char *() const;`
- declares an overloaded cast operator function for converting an object of class `MyClass` into a temporary `char *` object.
- The operator function is declared `const` because it does *not* modify the original object.