10.8 Case Study: A `Date` Class (cont.)

Date Class Postfix Increment Operator

- Overloading the postfix increment operator (defined in Fig. 10.7, lines 49–56) is trickier.
- To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object.
- So we'd like our postfix increment operator to operate the same way on a `Date` object.
- On entry to `operator++`, we save the current object (`*this`) in `temp` (line 51).
- Next, we call `helpIncrement` to increment the current `Date` object.
- Then, line 55 returns the unincremented copy of the object previously stored in `temp`.
- This function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it's declared exits.

Common Programming Error 10.1

Returning a reference (or a pointer) to a local variable is a common error for which most compilers will issue a warning.

10.9 Dynamic Memory Management

- You can control the *allocation* and *deallocation* of memory in a program for objects and for arrays of any built-in or user-defined type.
  - Known as dynamic memory management; performed with `new` and `delete`.
- You can use the `new` operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or built-in array at execution time.
- The object or built-in array is created in the free store (also called the heap)—*a region of memory assigned to each program for storing dynamically allocated objects.*
- Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns.
- You can return memory to the free store by using the `delete` operator to deallocate it.

10.9 Dynamic Memory Management (cont.)

Obtaining Dynamic Memory with new

- The `new` operator allocates storage of the proper size for an object of type `Time`, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the `new` operator (i.e., a `Time *`).
- If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception."

10.9 Dynamic Memory Management (cont.)

Releasing Dynamic Memory with `delete`

- To destroy a dynamically allocated object, use the `delete` operator as follows:
  - `delete ptr;`
- This statement first *calls the destructor for the object to which `ptr` points, then deallocates the memory associated with the object, returning the memory to the free store.*

Common Programming Error 10.2

Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."

Error-Prevention Tip 10.1

Do not delete memory that was not allocated by `new`. Doing so results in undefined behavior.

Error-Prevention Tip 10.2

After you delete a block of dynamically allocated memory be sure not to delete the same block again. One way to guard against this is to immediately set the pointer to `nullptr`. Deleting a `nullptr` has no effect.

10.9 Dynamic Memory Management (cont.)

Initializing Dynamic Memory

- You can provide an initializer for a newly created fundamental-type variable, as in
  - `double *ptr = new double( 3.14159 );`
- The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object.

10.9 Dynamic Memory Management (cont.)

Dynamically Allocating Built-In Arrays with new []

- You can also use the `new` operator to allocate built-in arrays dynamically.
- For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:
  - `int *gradesArray = new int[ 10 ]();`
- The parentheses following new `int[10]` value initialize the array's elements—fundamental numeric types are set to `0`, `bool`s are set to `false`, pointers are set to `nullptr` and class objects are initialized by their default constructors.
- A dynamically allocated array's size can be specified using *any* non-negative integral expression that can be evaluated at execution time.

10.9 Dynamic Memory Management (cont.)

C++11: Using a List Initializer with a Dynamically Allocated Built-In Array

- Prior to C++11, when allocating a built-in array of objects dynamically, you could not pass arguments to each object's constructor—each object was initialized by its default constructor. In C++11, you can use a list initializer to initialize the elements of a dynamically allocated built-in array, as in

```
int *gradesArray = new int[ 10 ]{};
```

- The empty set of braces as shown here indicates that default initialization should be used for each element—for fundamental types each element is set to 0.
- The braces may also contain a comma-separated list of initializers for the array's elements.

10.9 Dynamic Memory Management (cont.)

Releasing Dynamically Allocated Built-In Arrays with delete []

- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`
- *If the pointer points to a built-in array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory.*
- *Using `delete` or `[]` on a `nullptr` has no*

Common Programming Error 10.3

Using `delete` instead of `delete []` for built-in arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator `delete []`. Similarly, always delete memory allocated as an individual element with operator `delete`—the result of deleting a single object with operator `delete []` is undefined.

10.9 Dynamic Memory Management (cont.)

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

- C++11's new `unique_ptr` is a "smart pointer" for managing dynamically allocated memory.
- When a `unique_ptr` goes out of scope, its destructor automatically returns the managed memory to the free store.

10.10 Case Study: `Array` Class

- Pointer-based arrays have many problems, including:
  - A program can easily "walk off" either end of a built-in array, because *C++ does not check whether subscripts fall outside the range of the array.*
  - Built-in arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges- are not allowed.
  - An entire built-in array cannot be input or output at once.
  - Two built-in arrays cannot be meaningfully compared with equality or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the array's size must be passed as an additional argument.
  - One built-in array cannot be assigned to another with the assignment operator.