How to implement the Rule of Three with RAII

When you define a class in C++, if you don't explicitly define a destructor, a copy constructor, and a copy assignment operator, the compiler automatically generates default versions for you. Often, these default functions work fine. Sometimes, though, they don't. For example, if your class has a data member that's a pointer, the default copy constructor and copy assignment operator copy the pointer rather than the object that the pointer points to. This is called a *shallow copy*, and in most instances that's not what you want for this constructor and operator. When the default functions don't work, you need to write your own.

The *Rule of Three* states that if you need to define your own destructor, copy constructor, or copy assignment operator, you should define all of them as indicated by the table at the top of figure 17-12. In the previous figure, you learned that RAII requires you to define your own constructor and destructor to allocate and deallocate free store memory. Therefore, according to the Rule of Three, you should also define your own copy constructor and copy assignment operator when you use RAII.

The MyContainer class in this figure is an enhanced version of the MyContainer class from the previous figure that implements the Rule of Three. To start, the constructor and destructor shown here are the same as the constructor and destructor from the previous figure, except that they don't print messages to the console.

The copy constructor accepts a MyContainer object by constant reference. Then, the constructor uses this parameter to initialize itself. First, it uses the new keyword to allocate an array on the free store and stores the pointer to this array in the elements data member. Then, it makes a *deep copy* of the MyContainer object that's passed to it. To do that, it loops through the elements in the MyContainer parameter and copies those elements to the data member array.

This deep copy is critical. The default copy constructor generated by the compiler would just make a copy of the element pointer. In that case, both objects would have pointers that point to the same location on the heap. This would lead each object to call delete[] on the same heap memory when they went out of scope, which could cause memory corruption.

The copy assignment operator also accepts a MyContainer object by constant reference. Then, this function uses this parameter to update itself. First, it uses the new keyword to allocate an array on the free store and stores the pointer to this array in a temporary variable named temp. Then, it makes a deep copy of the MyContainer object that's passed to it and stores it in the temp variable. Next, it deallocates the memory that the elements pointer points to and updates that pointer so it points to the new array. Finally, it returns a reference to itself, which is what the default copy constructor that's generated by the compiler does.

This copy assignment operator could assign the pointer to the newly allocated memory directly to the elements data member and then perform the copy operation. However, it uses a temporary variable instead to provide exception safety. That way, if an exception occurs during the copy operation, the state of the container doesn't change. This avoids a situation where some, but not all, of the elements are changed and is generally considered a best practice.

The member functions required by the Rule of Three

Function	Description
destructor	Deallocates the memory allocated for the container's data.
copy constructor	Copies the data in the container passed to the constructor. Copied container is unchanged.
copy assignment operator	Copies the data in the container on the right side of the assignment operator (=). Copied container is unchanged.

The MyContainer class updated to implement the Rule of Three

```
class MyContainer {
private:
    int* elements = nullptr;
    int size = 10;
public:
    // constructor
    MyContainer() {
        elements = new int[size];
    }

    // destructor
    ~MyContainer() {
        delete[] elements;
    }

    // copy constructor
    MyContainer(const MyContainer& tocopy) {
        elements = new int[size];
        for (int i = 0; i < size; ++i) {
            elements[i] = tocopy.elements[i];    // deep copy
        }
    }

    // copy assignment operator
    MyContainer& operator= (const MyContainer& tocopy) {
        auto temp = new int[size];                // allocate new array
        for (int i = 0; i < size; ++i) {
            temp[i] = tocopy.elements[i];         // deep copy
        }
        delete[] elements;                        // delete old
        elements = temp;                          // assign new
        return *this;                             // return self-reference
    }
};
```

Description

- When you define a class, the compiler creates default copy constructor and copy assignment operator functions. In classes that use pointers, these functions copy the pointers, not the underlying objects. This is known as making a *shallow copy*.
- When you use RAII, you must define a constructor and destructor. With the *Rule of Three*, if you define (1) a destructor, you should also define (2) a copy constructor and (3) a copy assignment operator. With RAII, the copy constructor and assignment operator must copy the underlying objects, not the pointers. This is known as making a *deep copy*.

Figure 17-12 How to implement the Rule of Three with RAII

How to implement the Rule of Five with RAII

C++11 added a feature known as *move semantics* to the C++ language. This feature allows you to transfer data from one object to another rather than making copies of the data. As you might expect, moving data is more efficient than copying data, especially with large objects that are expensive to copy.

To implement move semantics in your own classes, you need to define a move constructor and a move assignment operator. These functions are described at the top of figure 17-13.

The addition of move semantics expands The Rule of Three to The Rule of Five. The *Rule of Five* states that if you need to define your own destructor, copy constructor, copy assignment operator, move constructor, or move assignment operator, you should define all of them.

As you've learned, RAII requires you to define your own destructor to deallocate free store memory. Therefore, according to the Rule of Five, if you want to include move semantics, you should also define your own copy constructor, copy assignment operator, move constructor, and move assignment operator when you use RAII.

The code example in this figure presents the MyContainer class from the last figure updated to implement the Rule of Five. The constructor, destructor, copy constructor, and copy assignment operator in this class are the same as in the previous figure. As a result, this figure doesn't show them again.

The move constructor accepts an *rvalue reference* to a MyContainer object. An rvalue reference identifies an object that can be moved. To declare a rvalue reference, you can use the *rvalue reference declarator* (`&&`).

Within the move constructor, the first statement copies the elements pointer of the parameter object to the elements pointer of the current object. At this point, both objects have pointers that point to the same memory address, which is the address where the array is stored. Then, the second statement sets the elements pointer of the parameter object to null so it no longer points to the array. This completes the move operation.

This move operation is more efficient than the copy operation shown in the previous figure. In that operation, the copy constructor allocated new memory and then copied each element in an array. Here, the move constructor doesn't need to allocate any new memory. Instead, it just reassigns the pointers that point to an existing array.

The move assignment operator also accepts an rvalue reference to a MyContainer object. Then, it uses this parameter object to update itself. To do that, it begins by using the delete[] keyword to deallocate the memory pointed to by the elements pointer of the current object. Next, it copies the elements pointer from the parameter object to the current object and sets the parameter object's pointer to null. Finally, it returns a reference to itself.

This move assignment operation is more efficient than the copy assignment operation in the previous figure. That's because this operation doesn't need to allocate new memory. Instead, it only needs to reassign the pointers to existing memory.

The additional member functions required by the Rule of Five

Function	Description
move constructor	Moves the data from the container passed to the constructor. Moved container is emptied.
move assignment operator	Moves the data from the container on the right side of the assignment operator (=). Moved container is emptied.

The MyContainer class updated to implement the Rule of Five

```
class MyContainer {
private:
    int* elements = nullptr;
    int size = 10;
public:
    // constructor is the same as in the previous figure
    // destructor is the same as in the previous figure
    // copy constructor is the same as in the previous figure
    // copy assignment operator is the same as in the previous figure

    // move constructor
    MyContainer(MyContainer&& tomove) {
        elements = tomove.elements;
        tomove.elements = nullptr;          // assign pointer to data
    }                                       // remove pointer to data

    // move assignment operator
    MyContainer& operator= (MyContainer&& tomove) {
        delete[] elements;                  // deallocate existing memory
        elements = tomove.elements;         // assign pointer to data
        tomove.elements = nullptr;          // remove pointer to data
        return *this;                       // return self-reference
    }
};
```

Description

- C++11 and later provide *move semantics* that enable you to move existing data. This is more efficient than copying existing data.
- The *rvalue reference declarator* (`&&`) indicates that the data of an object can be moved.
- To implement move semantics in your classes, you need to code a move constructor and a move assignment operator.
- When you use RAII, you must define a constructor and destructor. With the *Rule of Five*, if you define (1) a destructor, you should also define (2) a copy constructor, (3) a move constructor, (4) a copy assignment operator, and (5) a move assignment operator.

How to work with smart pointers

If you just want to do something simple like allocate an array in free store memory, it may not make sense to create all the member functions you need when using RAII, the Rule of Three, and the Rule of Five. Fortunately, with C++11 and later, you can use *smart pointers* instead. Unlike the *raw pointers* that you've worked with so far in this book, a smart pointer uses RAII under the hood to deallocate the free store memory automatically.

The top of figure 17-14 shows how to include the memory header file that contains the smart pointers. Then, it describes three of the smart pointers introduced in C++11. To start, unique_ptr objects can only point to one memory location. When this type of pointer goes out of scope, the system deallocates the memory. Because this type of pointer must be unique, it can't be copied, but it can be moved.

By contrast, shared_ptr objects can point to the same memory location. To do that, shared_ptr maintains a reference count. Then, when the reference count goes to zero, the system deallocates the memory. Because this type of pointer allows multiple pointers to point to the same location, it can be copied or moved.

You can create a weak_ptr object as a copy of a shared_ptr object. However, this type of pointer doesn't add to the reference count. Typically, weak_ptr objects are used with shared_ptr objects to avoid circular references that keep the reference count from ever reaching zero.

A unique_ptr has less overhead than a shared_ptr and is appropriate for most purposes. As a result, the examples in this figure show how to work with unique_ptr objects. However, the skills for working with shared_ptr and weak_ptr objects are similar.

The first example shows how to create a smart pointer that points to a single object or to an array of objects. When you define a smart pointer, you code the data type that the pointer refers to within angle brackets. To identify an array, you code square brackets after the data type. Then, you code the name of the pointer followed by a set of parentheses. Within these parentheses, you code the new keyword and the data type to allocate the free store memory. After that, you can work with the smart pointer just as you would a raw pointer, except that you can't use pointer arithmetic.

The second example shows how to use the make_unique() function, added in C++14, to create a unique_ptr to a single object or an array of objects. First, you code the auto keyword, the pointer name, and the assignment operator. Then, you call the make_unique() function with the data type within angle brackets. For an array, you code square brackets after the data type within the angle brackets. Then, you pass the number of elements as an argument to the function. The advantage of this function is that you can use the auto keyword, and you don't have to use the new keyword.

The last two examples work with a create_array() function that returns a unique_ptr to an array. This function works like the create_array() function presented in figure 17-10. However, that version returned a raw pointer to memory that needed to be deallocated manually. With this version, the smart pointer deallocates the memory automatically when the smart pointer to the array goes out of scope.

How to include the memory header file

```
#include <memory>
```

Three smart pointers in the memory header (C++11 and later)

Smart Pointer	Description
unique_ptr	Only one unique_ptr object can point to allocated memory. When unique_ptr is destroyed or goes out of scope, memory is deallocated.
shared_ptr	Multiple shared_ptr objects can point to allocated memory. Maintains a reference count and deallocates memory when count is zero.
weak_ptr	Created as a copy of a shared_ptr, but does not add to the reference count.

How to use unique_ptr to work with free store memory

Code that creates a smart pointer to an int

```
unique_ptr<int> ptr(new int);          // create smart pointer to int
*ptr = 4;                              // dereference and assign value
*ptr *= *ptr;                          // dereference and square value
cout << *ptr << endl;                  // displays 16
```

Code that creates a smart pointer to a built-in array

```
unique_ptr<int[]> arr(new int[10]);    // create smart pointer to array
for(int i = 0; i < 10; ++i) {          // set values of array elements
    arr[i] = i;
}
```

How to use the make_unique() function (C++14 and later)

```
auto ptr = make_unique<int>();         // create smart pointer to int
auto arr = make_unique<int[]>(10);     // create smart pointer to array
```

A create_array() function that returns a smart pointer to a built-in array

```
unique_ptr<int[]> create_array(unsigned int size) {
    auto arr = make_unique<int[]>(size);
    return arr;
}
```

Code that uses the create_array() function

```
int main() {
    unsigned int size = 0;
    cout << "Please enter the size of the array: ";
    cin >> size;

    auto arr = create_array(size);
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << ' ';
    }
} // the smart pointer automatically deallocates memory for the array
```

Description

- With C++ 11 and later, you can use *smart pointers* to manage memory automatically.
- The weak_ptr type is often used with the shared_ptr type to avoid circular references.
- There is also an auto_ptr type, but it was deprecated in C++11.

Figure 17-14 How to work with smart pointers