# Move Constructors and Move Assignment Operators (C++)

Article • 08/03/2021 • 6 minutes to read

This topic describes how to write a *move constructor* and a move assignment operator for a C++ class. A move constructor enables the resources owned by an rvalue object to be moved into an lvalue without copying. For more information about move semantics, see Rvalue Reference Declarator: &&.

This topic builds upon the following C++ class, `MemoryBlock`, which manages a memory buffer.

C++

```cpp
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

   // Simple constructor that initializes the resource.
   explicit MemoryBlock(size_t length)
      : _length(length)
      , _data(new int[length])
   {
      std::cout << "In MemoryBlock(size_t). length = "
                << _length << "." << std::endl;
   }

   // Destructor.
   ~MemoryBlock()
   {
      std::cout << "In ~MemoryBlock(). length = "
                << _length << ".";

      if (_data != nullptr)
      {
         std::cout << " Deleting resource.";
         // Delete the resource.
         delete[] _data;
      }

      std::cout << std::endl;
   }
   // Copy constructor.
   MemoryBlock(const MemoryBlock& other)
      : _length(other._length)
      , _data(new int[other._length])
   {
      std::cout << "In MemoryBlock(const MemoryBlock&). length = "
                << other._length << ". Copying resource." << std::endl;

      std::copy(other._data, other._data + _length, _data);
   }

   // Copy assignment operator.
   MemoryBlock& operator=(const MemoryBlock& other)
   {
      std::cout << "In operator=(const MemoryBlock&). length = "
                << other._length << ". Copying resource." << std::endl;

      if (this != &other)
      {
         // Free the existing resource.
         delete[] _data;

         _length = other._length;
         _data = new int[_length];
         std::copy(other._data, other._data + _length, _data);
      }
      return *this;
   }

   // Retrieves the length of the data resource.
   size_t Length() const
   {
      return _length;
   }

private:
   size_t _length; // The length of the resource.
   int* _data; // The resource.
};
```

The following procedures describe how to write a move constructor and a move assignment operator for the example C++ class.

## To create a move constructor for a C++ class

1. Define an empty constructor method that takes an rvalue reference to the class type as its parameter, as demonstrated in the following example:

   C++

```
MemoryBlock(MemoryBlock&& other)
   : _data(nullptr)
   , _length(0)
{
}
```

2. In the move constructor, assign the class data members from the source object to the object that is being constructed:

C++

```
_data = other._data;
_length = other._length;
```

3. Assign the data members of the source object to default values. This prevents the destructor from freeing resources (such as memory) multiple times:

C++

```
other._data = nullptr;
other._length = 0;
```

## To create a move assignment operator for a C++ class

1. Define an empty assignment operator that takes an rvalue reference to the class type as its parameter and returns a reference to the class type, as demonstrated in the following example:

C++

```
MemoryBlock& operator=(MemoryBlock&& other)
{
}
```

2. In the move assignment operator, add a conditional statement that performs no operation if you try to assign the object to itself.

C++

```
if (this != &other)
{
}
```

3. In the conditional statement, free any resources (such as memory) from the object that is being assigned to.

   The following example frees the `_data` member from the object that is being assigned to:

   C++

   ```cpp
   // Free the existing resource.
   delete[] _data;
   ```

   Follow steps 2 and 3 in the first procedure to transfer the data members from the source object to the object that is being constructed:

   C++

   ```cpp
   // Copy the data pointer and its length from the
   // source object.
   _data = other._data;
   _length = other._length;

   // Release the data pointer from the source object so that
   // the destructor does not free the memory multiple times.
   other._data = nullptr;
   other._length = 0;
   ```

4. Return a reference to the current object, as shown in the following example:

   C++

   ```cpp
   return *this;
   ```

## Example: Complete move constructor and assignment operator

The following example shows the complete move constructor and move assignment operator for the `MemoryBlock` class:

C++

```cpp
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
   : _data(nullptr)
   , _length(0)
{
```

```
      std::cout << "In MemoryBlock(MemoryBlock&&). length = "
                << other._length << ". Moving resource." << std::endl;

      // Copy the data pointer and its length from the
      // source object.
      _data = other._data;
      _length = other._length;

      // Release the data pointer from the source object so that
      // the destructor does not free the memory multiple times.
      other._data = nullptr;
      other._length = 0;
   }

   // Move assignment operator.
   MemoryBlock& operator=(MemoryBlock&& other) noexcept
   {
      std::cout << "In operator=(MemoryBlock&&). length = "
                << other._length << "." << std::endl;

      if (this != &other)
      {
         // Free the existing resource.
         delete[] _data;

         // Copy the data pointer and its length from the
         // source object.
         _data = other._data;
         _length = other._length;

         // Release the data pointer from the source object so that
         // the destructor does not free the memory multiple times.
         other._data = nullptr;
         other._length = 0;
      }
      return *this;
   }
```

## Example Use move semantics to improve performance

The following example shows how move semantics can improve the performance of your applications. The example adds two elements to a vector object and then inserts a new element between the two existing elements. The `vector` class uses move semantics to perform the insertion operation efficiently by moving the elements of the vector instead of copying them.

C++

```
// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
   // Create a vector object and add a few elements to it.
   vector<MemoryBlock> v;
   v.push_back(MemoryBlock(25));
   v.push_back(MemoryBlock(75));

   // Insert a new element into the second position of the vector.
   v.insert(v.begin() + 1, MemoryBlock(50));
}
```

This example produces the following output:

Output

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.
```

Before Visual Studio 2010, this example produced the following output:

Output

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.
In ~MemoryBlock(). length = 75. Deleting resource.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In operator=(const MemoryBlock&). length = 75. Copying resource.
In operator=(const MemoryBlock&). length = 50. Copying resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.
```

The version of this example that uses move semantics is more efficient than the version that does not use move semantics because it performs fewer copy, memory allocation, and memory deallocation operations.

## Robust Programming

To prevent resource leaks, always free resources (such as memory, file handles, and sockets) in the move assignment operator.

To prevent the unrecoverable destruction of resources, properly handle self-assignment in the move assignment operator.

If you provide both a move constructor and a move assignment operator for your class, you can eliminate redundant code by writing the move constructor to call the move assignment operator. The following example shows a revised version of the move constructor that calls the move assignment operator:

C++

```cpp
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
   : _data(nullptr)
   , _length(0)
{
   *this = std::move(other);
}
```

The std::move function converts the lvalue `other` to an rvalue.

## See also

Rvalue Reference Declarator: &&

std::move