Analysis of Data Copy Mechanism

Memory management in modern languages

Dawid Błaszczyk
International Journal of
Computer Science Research

Filip Kolasa
International Journal of
Computer Science Research

Abstract

In modern times, when writing programs, one does not have to think about many issues integral to creating them in the past. Such possibilities are provided by modern programming languages containing several different types of tools that allow developers to go directly to the implementation of a given issue. One of the aspects they hide is memory management, which is often performed automatically by the so-called *"Garbage Collector"*. These algorithms remove objects to which the program no longer has access. Introducing such general simplifications is associated with the lack of need to familiarize oneself with the conventions describing the operation of memory. In turn, a lack of familiarity with them may result in the emergence of a problem described in this article.

Keywords: analysis, data, copy mechanism, memory management, modern languages

1 Introduction

1.1 The Problem

Modern programming languages often hide how memory is managed to a large extent. When writing a program, knowledge of the conventions related to it is not required, which allows direct focus on the implemented issue. However, this is associated with a specific problem, which was decided to be presented in the example of the Java language.

A program was created to demonstrate the problem. Its goal is to create a list of digits from 1 to 3 in ascending order, copy the list, and then remove the last element from the copied list, add the digit 1 to the end of the copied list, and change the value of the first digit of the copied list to 3. The expected result of the program is two lists with digits from 1 to 3, with the original list written in ascending order and the copied list written in descending order.

In addition, Java introduces the concept of immutable and mutable classes that define the ability to modify the state of an object after its creation. Objects of immutable classes cannot change their state, while objects of mutable classes can change their state. Therefore, the program will be executed twice for both types of these classes.

1.2 Code

1.2.1 Immutable example

An example of an immutable class is the *"Integer"*, a wrapper class for the primitive type *"int"*. This class allows it to perform various operations on the number it represents. An object of this class can be created using a constructor that takes the primitive type *"int"* (see lst. 1).

Listing 1: Immutable *"Integer"* class.

```
Integer value = new Integer(5);

// value = new Integer(10);
value = 10;
```

The variable *"value"* is a reference to an object of the class *"Integer"*. Since the class *"Integer"* is immutable, it is impossible to change its store value. Using the assignment operator, a new class object, *"Integer"*, is created with the given value. Therefore, the variable *"value"* becomes a reference to the newly created object.

Running the test program with the immutable *"Integer"* class produces the expected result (see tab. 1).

Table 1: Immutable test result.

List elements	
1	3
2	2
3	1

1.2.2 Mutable example

The vast majority of classes in Java are mutable. However, to present an analogous example to the immutable class, a wrapper of the *"Integer"* class, a mutable class called *"MutableInteger"*, has been implemented (see lst. 2).

Listing 2: *"MutableInteger"* class.

```
public class MutableInteger {

    private Integer value;

    public MutableInteger(
   Integer value) {
        this.value = value;
    }

    public Integer getValue()
   {
        return value;
    }

    public void setValue(
   Integer value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return value.toString
   ();
    }
}
```

The *"MutableInteger"* class provides a simple interface for setting and getting the value of the *"value"* field, which is an *"Integer"* object. Additionally, when using the print function, the *"toString"* method has been overridden to display the number represented by the *"value"* field. The constructor that takes an *"MutableInteger"* object can be used to create a *"Integer"* object.

Running the test program with the mutable class *"MutableInteger"* did not produce the expected result (see tab. 2).

Table 2: Mutable test result.

List elements	
3	3
2	2
3	1

Running the test program with the mutable class *"MutableInteger"* modifies the object in the original list. This means that changing the value of the copied list object affects the value of the original list object. This situation can only occur in the case of sharing resources by both lists, which were copied using the copy constructor, so it should not occur.

In order to understand the problem presented in Java, one must understand the basics of memory management behind dynamically allocated containers. A learning aid in this regard will be an example implemented in C++, which allows low-level memory access.

2 Lower-level insight

2.1 *"DynamicArray"* class

Unlike Java, the C++ language allows low-level access to memory. This is associated with many advantages that allow programming in a way usually unavailable in modern programming languages—an example of how containers with dynamically allocated resources work will be presented using these possibilities.

The *"DynamicArray"* class will be a container for storing *"int"* type elements. It will be based on a dynamically allocated array in the form of an *"int* Data"* pointer and a variable tracking the current size of the array *"uint64_t Size"*. The container interface will allow for changing the size of the array, accessing its elements, and the ability to remove all elements from the array.

2.2 Implementation

2.2.1 Memory safety

The class has been implemented in a way that centralizes memory management. This is crucial to minimize the risk of introducing errors related to it. Each memory allocation must be deallocated before the program exits; otherwise, a *"Memory Leak"* error occurs. This error describes a state in which, despite the program exiting, not all the memory used by it has been freed. The methods directly responsible for memory management are *"Reallocate"* and *"Clear"*.

The *"Reallocate"* method (see lst. 3) is responsible for reallocating a new array of a given size. Depending on this size, the contents of the current array are copied to the new array using the *"CopyMemory"* helper method.

Listing 3: Method *"Reallocate"*.

```
void Reallocate(uint64_t SIZE)
{
    int* NewData = new int[SIZE];
    if (Size > SIZE)
    {
        CopyMemory(NewData, Data, SIZE);
    }
    else
    {
        CopyMemory(NewData, Data, Size);
    }
    delete[] Data;
    Data = NewData;
    Size = SIZE;
}
```

The *"CopyMemory"* helper method (see lst. 4) copies the contents of the source array to the destination array from the beginning of the array to the given size.

Listing 4: Method *"CopyMemory"*.

```
void CopyMemory(int* DESTINATION, int* SOURCE, uint64_t SIZE)
{
    for (uint64_t i = 0; i < SIZE; ++i)
    {
        DESTINATION[i] = SOURCE[i];
    }
}
```

The *"Clear"* method (see lst. 5) is responsible for memory deallocation.

Listing 5: Method *"Clear"*.

```
void Clear()
{
    delete[] Data;
    Data = nullptr;
    Size = 0;
}
```

This is the only memory management method that is part of the class's public interface and thus allows all elements to be removed from the array.

2.2.2 Interface

The *"DynamicArray"* class implements two constructors (see lst. 6).

Listing 6: Constructors.

```
DynamicArray()
    : Data(nullptr), Size(0) {
    }

DynamicArray(uint64_t SIZE)
    : Data(nullptr), Size(0)
{
    Resize(SIZE);
}
```

The first is its custom default constructor that initializes all fields to the assumed default values. The second one, apart from initializing the fields, also allows to set the size of the array using the *"Resize"* method.

The *"Resize"* method (see lst. 7) allows for the change of the size of an array.

Listing 7: Method *"Resize"*.

```
void Resize(uint64_t SIZE)
{
    if (SIZE == 0)
    {
        Clear();
    }
    else
    {
        Reallocate(SIZE);
    }
}
```

Depending on the expected new array size specified, either the *"Clear"* or *"Reallocate"* method is called with the given array size.

In order to allow access to array elements, the overloaded *"[]"* operator (see lst. 8) is included, with both cases for regular objects and objects with the *"const"* modifier being considered.

Listing 8: Operator[].

```
int& operator[](uint64_t INDEX
    )
{
    return (Data[INDEX]);
}

const int& operator[](uint64_t
    INDEX) const
{
    return (Data[INDEX]);
}
```

This operator is used to access elements of traditional arrays.

At the end of an object's lifetime, its custom destructor (see lst. 9) calls the *"Clear"* method to deallocate the array.

Listing 9: Destructor.

```
~DynamicArray()
{
    Clear();
}
```

2.2.3 Test program

A test program was created to test the *"DynamicArray"* class. Its task is to create a *"DynamicArray"* class object with an array of 3 elements, make a copy of this object, and then display the memory addresses of both objects and their arrays. The expected effect of the program is to create a copy of the object containing the same elements in the array as the original object but with different memory addresses relative to the original object.

Running the program has the same effect as running a Java program (see tab. 3).

Table 3: *"DynamicArray"* test result.

Object addresses	
0x61fc70	0x61fc80
Array addresses	
0xa11b90	0xa11b90
0xa11b94	0xa11b94
0xa11b98	0xa11b98

Despite creating a copy of the object, the new object's array is the same as in the original object. When using a ready container that performs the same task, e.g., *"std::vector"* contained within the *"Standard Template Library"* that is part of the C++ language, the program's effect is consistent with expectations (see tab. 4).

Table 4: *"std::vector"* test result.

Object addresses	
0x61fd90	0x61fdb0
Array addresses	
0xae1b90	0xae1bb0
0xae1b94	0xae1bb4
0xae1b98	0xae1bb8

The difference in the behavior of both classes, indicating the undesirable effect when using the *"DynamicArray"* class, is that the *"DynamicArray"* class is implemented incorrectly. The problem, however, is not in the currently written code but in the lack of code for its custom implementation of copy semantics and move semantics.

2.2.4 Copy semantic

Copy semantics is a description of the behavior of the object copying system. This process, by default, consists of copying the values of all fields of the source object to the corresponding fields of the target object of the same class. Copying field values is performed according to the copy semantics appropriate to the field type. As a result of copying, both objects are valid and contain the same field values.

For classes that contain raw pointers to dynamically allocated memory by their objects, default copy semantics must be modified. In this case, the pointer value is copied, not the allocated memory value. A custom copy constructor and copy assignment operator must be implemented to modify the default copy behavior.

To avoid code duplication, creating a private helper method called *"CopyFrom"* is recommended, which will be responsible for the actual copying process. In the case of pointers that require a behavior change, the whole thing can be generalized to using memory deallocation in the target object, memory allocation in the target object with the size of the memory in the source object, and copying via the assignment operator of the source object's memory value to the target object's memory. The remaining fields should be copied via the assignment operator.

The following listing shows the *"CopyFrom"* (see lst. 10) method implemented within the example class.

Listing 10: Method *"CopyFrom"*.

```
void CopyFrom(const DynamicArray& OTHER)
{
    Clear();
    Data = new int[OTHER.Size];
    Size = OTHER.Size;
    CopyMemory(Data, OTHER.Data, OTHER.Size);
}
```

The following listing shows the copy constructor (see lst. 11) implemented within the example class.

Listing 11: Copy constructor.

```
DynamicArray(const DynamicArray& OTHER)
    : Size(0),
      Data(nullptr)
{
    CopyFrom(OTHER);
}
```

The following listing shows the copy assignment operator (see lst. 12) implemented within an example class.

Listing 12: Copy assignment.

```
DynamicArray& operator=(const DynamicArray& OTHER)
{
    if (this == &OTHER)
    {
        return (*this);
    }
    CopyFrom(OTHER);
    return (*this);
}
```

2.2.5 Move semantic

Move semantics is a description of the behavior of an object transfer system. This process, by default, transfers ownership of all field values of a source object to the corresponding fields of a target object of the same class by copying them. Transferring ownership of field values is done according to the transfer semantics appropriate to the field type. As a result of the transfer, only the target object is valid and contains the original field values of the source object, while the source object potentially loses its field values and should no longer be used.

For classes that contain raw pointers to dynamically allocated memory by their objects, default move semantics must be modified. In this case, the pointer value is copied, and ownership of the allocated memory is not transferred. A custom move constructor and move assignment operator must be implemented to modify the default move behavior.

To avoid code duplication, creating a private helper method called *"MoveFrom"* is recommended, which will be responsible for the actual moving process. In pointers requiring a behavior change, the whole thing can be generalized to assigning the source object's value to the target object and assigning the *"nullptr"* value to the source object. The remaining fields should be copied via the assignment operator, and if necessary, their values should be set to the default in the source object.

The following listing shows the *"MoveFrom"* (see lst. 13) method implemented within the example class.

Listing 13: Method *"MoveFrom"*.

```
void MoveFrom(DynamicArray&
    OTHER)
{
  Clear();
  Size = OTHER.Size;
  Data = OTHER.Data;
  OTHER.Size = 0;
  OTHER.Data = nullptr;
}
```

The following listing shows the move constructor (see lst. 14) implemented within the example class.

Listing 14: Move constructor.

```
DynamicArray(DynamicArray&&
    OTHER) noexcept
    : Size(0),
      Data(nullptr)
{
    MoveFrom(OTHER);
}
```

The following listing shows the move assignment operator (see lst. 15) implemented within an example class.

Listing 15: Move assignment.

```
DynamicArray& operator=(
    DynamicArray&& OTHER)
    noexcept
{
    if (this == &OTHER)
    {
        return (*this);
    }
    MoveFrom(OTHER);
    return (*this);
}
```

2.3 Result

Modifying the *"DynamicArray"* class with a custom implementation of copy semantics and move semantics results in the expected result of the test program (see tab. 5).

Table 5: *"DynamicArray"* test result 2.

Object addresses	
0x61fc70	0x61fc80
Array addresses	
0x871b90	0x871bb0
0x871b94	0x871bb4
0x871b98	0x871bb8

The lack of its custom implementation of copy semantics and move semantics for classes containing raw pointers to dynamically allocated memory by their objects is a bug in the C++ language. Apart from the unintended sharing of resources, this leads to the so-called *"Segmentation Fault"* at various stages of the program's operation. This error describes a state in which the application tries to read or write memory that the program has not allocated, thus leading to a violation of memory safety. It is therefore recommended to follow the principle called *"The rule of five"*, which states that if a class implements its methods related to copy semantics, its methods related to move semantics, or its destructor, all of the above five methods should probably be implemented.

Unlike C++, however, such an action is not an error in Java because the *"Garbage Collector"* is responsible for memory management. Various algorithms implement the mechanics of memory cleaning, but the most popular are tracking the number of references and tracking the chain of references.

The reference count tracking algorithm counts the number of references to a given object. When the reference count equals 0, the object is no longer available to the program, and the *"Garbage Collector"* mechanism removes the object in such a case. In a simple implementation of this algorithm, there is a problem with self-references because the reference count will never be equal to 0.

Unlike the reference count tracking algorithm, the reference chain tracking algorithm does not have this problem. It consists of tracking the references of all objects from the root object and marking them as objects that the program has access to. Objects that are not marked in this way are objects that the program no longer has access to.

Java behaves like the original example of the *"DynamicArray"* class by default without implementing its custom copy semantic and move semantic. The difference is that when an object that shares memory with another object is deleted, the memory itself is not deleted because there is still a reference pointing to that memory. The *"Garbage Collector"* only deletes the memory when there is no longer a chain of references that would allow reaching the memory according to the chain of reference tracking algorithm.

3 Summary

3.1 The Solution

Using a C++ example, it is possible to understand the problem that occurred in Java. When the list copy constructor is called, a copy of only the list object is created, not the additional elements contained in it. As a result, both lists contain references to the same objects (see fig. 1).

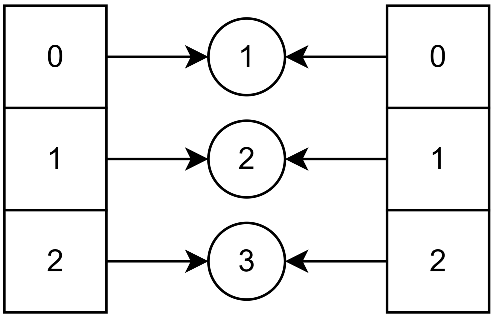

Figure 1: Shallow copy example.

By then modifying the copied list by removing its elements, references to the shared objects are removed, and by adding elements, new references to new, no longer shared objects are added. By modifying list elements, shared objects are modified, affecting both lists (see fig. 2).

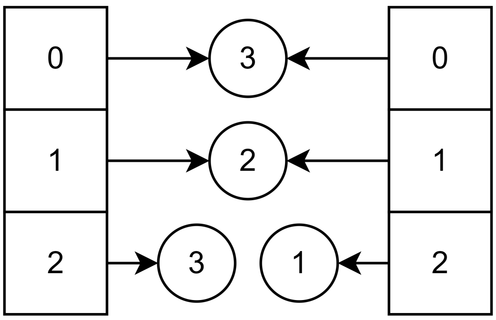

Figure 2: Sharing memory example 1.

The situation is slightly different when modifying elements of immutable classes. Since objects of these classes cannot change their state, new objects are created, and the whole thing comes down to removing and adding list elements. However, this does not change the fact that both lists potentially continue sharing resources, which is undesirable (see fig. 3).

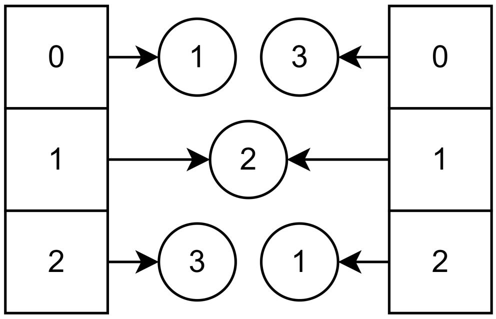

Figure 3: Sharing memory example 2.

In order to solve the problem, some mechanisms, like those presented in C++, must be used. The simplest way to implement copy semantic and move semantic is impossible due to the use of a ready container and the need for low-level memory access. Another possibility is to extend the class *"MutableInteger"* with the interface *"Copyable"*, whose implementation of the *"clone"* method by creating a new object and copying the contents of the original object should be a solution. Unfortunately, this method also does not work due to various problems with the *"Copyable"* interface at the language implementation level. The only way is to create a custom method that copies the list and its contents.

3.2 Code

A new list object must be created to copy the original list. Then, a copy of each object in the original list must be made and inserted into the new list. The following list shows an example method for making a copy of the original list and its contents (see lst. 16).

Listing 16: Method *"deepCopy"*.

```
public ArrayList<MutableInteger> deepCopy(ArrayList<MutableInteger> list) {

    ArrayList<MutableInteger> newList = new ArrayList();

    for (MutableInteger element : list) {
        newList.add(new MutableInteger(element.getValue()));
    }

    return newList;
}
```

Only a copy made in this way guarantees no sharing of resources between lists (see fig. 4).

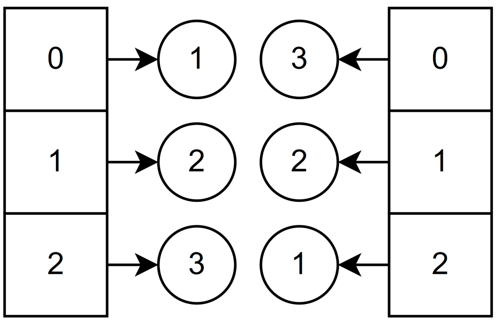

Figure 4: Deep copy example.

Running the test program with the mutable class *"MutableInteger"* and a new list copy produces the expected result (see tab. 6).

Table 6: Mutable test result 2.

List elements	
1	3
2	2
3	1

At this stage, however, it is necessary to consider the whole problem. The apparent bug in C++ that causes many memory problems is not a bug in modern programming languages. In place of the bug, two new concepts appear: shallow copy and deep copy. A shallow copy is a copy that refers only to the object being copied without paying attention to its contents, and a deep copy is a copy that takes into account the contents of the copied object. This is a natural part of the behavior of modern programming languages.

Therefore, the problem's origin can be found in the modern programming languages themselves and the simplifications they introduce. However, these simplifications constitute what these programming languages are and their purpose. If it were not for them, the process of creating a large part of the applications currently used would be significantly prolonged, so they cannot be perceived as a problem. The real problem, the solution of which they are, is C++. If this language were created with tools adapted to modern requirements, there would never be a need to create new simplified programming languages.