A practical comparison of the Java and C++ resource management facilities

Markus Winand

2003-02-08

Abstract

This paper is a comparative description of the resource management features of the Java and C++ programming languages. Although there are already many "comparisons" of those two languages, it's the opinion of the author that most of them are not very professional by means of objectivity. I tried really hard to be objective (or at least more objective) and show the strength and weaknesses of both languages.

The paper is divided into four sections, the first discusses the memory management[1] and focuses on exception safety, the second compares the Java finalizer and the C++ destructor. The third section discusses Java finally blocks, and how C++ avoids the requirement for those. The last section puts the pieces together and implements a commonly used structure in both languages.

[1] It has to be mentioned that this paper does not cover the `operator new` overload possibilities of the C++ programming language.

1 Memory management

1.1 Java concepts

The Java programming language has a built in garbage collection system. Based on the reachability of an object (are there still references in the program to this object) the JVM[2] starts the finalization process. [GJSB00, §12.6.1] includes the full state diagram of all eight possible object states. The two key states of objects are the reachability and the finalization state. Both of them have three possible values. Objects may be reachable, finalizer-reachable or unreachable in terms of reachability or unfinalized, finalizeable or finalized in terms of finalization. Both states together result in a 3-by-3 matrix including all possible object states, where one state (unreachable and finalizeable) is impossible. This complex model is required to make garbage collection of circular referenced objects possible. This is an important feature to allow leak free garbage collection for long running processes where circular references occur (e.g. object `A` references to `B` and vice versa). A pure reference count based garbage collector is not able of collecting circular references, many of them require the developer to manually break the loop[3].

The Java garbage collector (like most others) does not specify when a finalizeable object becomes finalized. The garbage collector does not guarantee to finalize the objects in the order they became finalizeable (which of course is impossible for circular references).

The benefit of the garbage collection is that it reduces the probability of memory leaks. The responsibility to release unused memory has been moved from the developer to the garbage collection, but there are still ways to leak memory[4]. As a result of the introduction of the garbage collector, the developer has no way to explicitly delete an object. The only thing that can be done (and must be done sometimes to avoid unused data stay referenced) is to set the reference to an unused object to `null`. That way the object becomes finalizer-reachable and will be garbage collected eventually (as long as there are no other references).

[2] Java Virtual Machine

[3] The perl garbage collector has this limitation.

[4] This can happen if a resource stays referenced by accident. This most probably happens when using collections (containers).

1.2 C++ concepts

The C++ programming language does, per default, not use garbage collection. But in contrast to the Java language, there are three different locations where objects can be placed[5]:

- In the static memory
- On the stack (local store)
- On the heap (free store)

The following sections will explain the differences between these locations, how they can be combined and which benefits can be gained from this.

1.2.1 Static objects

Objects in the static memory are objects which are created on start of the program and will be available until the program terminates. Global and namespace variables are always placed in the static memory, as well as class or function variables[6] with the `static` modifier.

Static objects are constructed only once at program startup and deleted on normal program termination[7]. The memory requirement is handled by the compiler.

1.2.2 Objects on the stack

Objects located on the stack are fully managed by the compiler. There is no need to manually allocate memory of delete them when they are no longer used.

To allow the compiler to take care of this details there are some limitations: only static sized objects (objects for which the size is known at compile time)[8] and objects whose lifetime does

[5] See [Str97, §C.9]

[6] Static function variables are not created on program startup, but the first time their declaration is encountered during execution of the program.

[7] Static objects are not destructed if the program terminates because `_exit`, `abort` or similar functions are called. Even if you are not using this functions, it might happen indirectly if an unexpected exception [Str97, §14.6.2] or uncaught exception [Str97, §14.7] happens.

[8] In fact every object has a static size which is known at compile time (determined using the `sizeof` operator). As a result, every object can be placed on the stack. But this does not mean that all memory requirements for this object can be satisfied by the stack: if you place a `string` on the stack, the implementation of the `string` would need to allocate memory from the heap to fulfill it's duty.

not exceed the block they are declared in, can be placed on the stack.

Listing 1 demonstrates two objects (integer values) placed on the stack. The `x` variable is

Listing 1: C++ Allocation on the stack

```
1  void StackAlloc()
2  {
3    int x(0);
4    {
5      x = 3;
6      int y(x);
7      ++y;
8    }
9    x++;
10 }
```

declared in line 3 and exists until the block in which it was declared ends (line 10). Between it's declaration and the block end in line 10 the variable can be used. The `y` variable is constructed in line 6[9] and becomes destroyed in line 8.

As you can see, the construction of the objects is done explicit, but the destruction is done automatically by the compiler. The compiler will call the destructor of the objects which went out of scope in reverse order of their construction. If a constructor throws an exception, the corresponding destructor will not be called (only successfully constructed objects can be destructed).

It is an important fact that objects on the stack can not leak[10].

1.2.3 Objects on the heap

Objects on the heap are dynamically allocated at runtime. The C++ language provides the `new` operator to allocate dynamic memory[11]. A object created with the `new` operator exists until it is explicitly destroyed using the `delete` operator.

Listing 2 demonstrates two integers which are allocated on the heap[12]. Besides the already mentioned `new` and `delete` operators, we notice

Listing 2: C++ Allocation on the heap

```
1  void HeapAlloc()
2  {
3    int* x = new int(0);
4    int* z;
5    {
6      *x = 3;
7      int& y = *new int(*x);
8      y++;
9      z = &y;
10   }
11   (*x)++;
12   delete x;
13   ++(*z);
14   delete z;
15 }
```

that the integers are used through pointers and references[13]. This is a result of the fact that the `new` and `delete` operators operate on pointers.

The first object on the heap is constructed in line 3, a pointer to the newly created integer is stored in the `x` variable. This object exists until it gets deleted in line 12. In the meantime it can be used as pointer, or it's value can be accessed and manipulated using the unary `*` operator. Another way is to use a reference as shown in line 7. In that case `y` becomes an alias for the integer created on the heap. The `y` variable can be used in the same way as an object on the stack can be used (as long as it has not been deleted).

The integer allocated in line 7 can also be accessed outside the block where it was allocated (line 13)[14]. Further the objects can be destroyed in another order than they were created.

The cost of this flexibility is that that you have to manage everything yourself. The compiler does not delete the object for you and will allow you to use a pointer to an object which has

[9] Other than C the C++ language allows local variables (objects on the stack) to be declared in the middle of a block (C requires all declarations to be at the start of the block).

[10] This is even true if an exception occurs, more details later

[11] In fact there is a set of operators which are related to dynamic memory management. [Str97, §19.4.5] lists six variations of `operator new` and a matching `operator delete` for each. This paper will only use the most basic variant. For details about the other variants (nothrow, array, and placement) have a look at [Str97].

[12] In fact we see the use of two integers, two pointers (`x` and `z`) and one reference (`y`). The integers are located on the heap, the pointers and references are located on the stack. You always need a pointer or reference to access an object on the heap.

[13] See [Str97, §5.5]

[14] Note that the use of the `y` variable is not possible outside the inner block, this is because the `y` variable itself (the reference to the integer) is located on the stack and will be deleted when the block ends in line 10.

The object which is referenced by this local variable is not affected, but it is an important fact that the references or pointers which are used to access the object on the heap are located on the stack.

To allow the access outside this block, we have copied the pointer to this object to the `z` variable which is accessible in the outer block.

already been deleted[15]. But the most problematic aspect of this is that you have to manage the cleanup of dynamically allocated memory even in case of exceptions[16]. To address those aspects of dynamically allocated memory C++ provides the following techniques:

- Avoid the requirement for manually managed dynamic memory
- Using smart pointer types
- Possibility to use custom garbage collectors

The last point will not be covered in detail here, in short there are systems which implement a garbage collection for the C++ programming language similar to Java. It has to be mentioned that many of those garbage collectors have a common limitation: they can not destruct the object being collected. This means the destructor of the objects will not be called. In other words, the garbage collector does not destroy the object properly, they just re-use the memory occupied by this object. Another description for this technique is the "infinite memory" model, which means that the only resource which is correctly collected is the memory.

The first point is addressed with the STL[17] which provides many generic class implementations for common requirements. The STL is part of the C++ specification and is therefore available on all C++ platforms. The majority of STL classes implements some type of container[18] like linked lists, hash tables and strings. Most of the time the STL can handle the dynamic memory for you and eliminate the requirement to handle it manually.

The second point, smart pointer types, will be covered in the following sections.

2 Destructor vs. finalizer

Both languages allow to implement some kind of cleanup code which is executed before the objects life time ends. The only difference is that it is undefined when the Java finalizer will be called (as already mentioned in 1.1) but it is clearly defined when the C++ destructor is executed.

Those cleanup methods are usually required to release some resources which where bound in the constructor. If you would, for example, allocate dynamic memory in the constructor of a C++ object, you need to deallocate this memory if it is no longer used. Listing 3 demonstrates such a C++ class with destructor.[19] The constructor does initialize the private member variable `_p`[20] with a pointer to a dynamically allocated integer (line 3). The object could use this integer during it's life time, but if the `CPPdeallocate` object gets deleted its destructor will be called which will delete the object where `_p` points to (line 9).

Listing 3: C++ destructor example

```
1  class CPPdeallocate {
2    public:
3      CPPdeallocate(): _p(new int())
4      {
5      };
6
7      ~CPPdeallocate()
8      {
9        delete _p;
10     }
11
12   private:
13     int* _p;
14 };
```

The implementation of a similar example in Java would not require a finalizer since the garbage collector would take care about this. For this reason the Java finalizers are less frequently used than C++ destructors.

The drawback of the Java solution is that it does well for memory, but what about other resources? For example a file-handle, open socket or similar. With C++ you can just use the same technique, but for Java the use of the finalizer is usually not sufficient, since it is undefined when it's called[21]. Listing 4 can be used to demonstrate this. The `work` method constructs a new resource, since the reference to this resource gets lost when the `work` method exits, the resource could be released. The `ptime` and `sleepwell` methods are used for reporting and delaying. The `main` method does call the `work` method, sleeps two seconds and then calls the garbage

[15] e.g. if you would use `*x` after line 12

[16] If line 7 would cause an exception, the object allocated in line 3 would be lost. The memory occupied by this object would leak.

[17] **S**tandard **T**emplate **L**ibrary

[18] Also known as *collection* in Java terms

[19] C++ requires the destructor to be named like the class itself with a leading tilde.

[20] This paper uses the convention to start non public member variables with a leading underscore.

[21] See [GJSB00, §12.6].

Listing 4: Java finalizer example

```
1  public class JavaFinalize {
2    private static void work() {
3      myresource t = new myresource();
4    }
5
6    public static void ptime(String info) {
7      System.out.println(info + " : " +
8        System.currentTimeMillis());
9    }
10
11   private static void sleepwell(long m) {
12     try {
13       Thread.sleep(m);
14     } catch (Exception e) { }
15   }
16
17   public static void main(String args[]) {
18     ptime("Start      ");
19     work();
20     ptime("sleep      ");
21
22     sleepwell(2000);
23
24     ptime("sleep done ");
25     System.gc();
26     ptime("End        ");
27   }
28 }
29
30 class myresource {
31   protected void finalize() throws Throwable
32   {
33     JavaFinalize.ptime("finalizer  ");
34     super.finalize();
35   }
36 }
```

collector[22]. For all steps the system timestamp will be printed. The `myresource` class in the end does only implement a finalizer that prints the timestamp as well. As required in [GJSB00, §12.6], each finalizer has to propagate the finalization to its super class manually.

If you run this program, you will notice that the finalizer gets called after the sleep has been done. If you would sleep longer, the stale object would be deleted even later. If you wonder what happens if you remove the explicit call to the garbage collector, give it a try. There is not even a guarantee that the finalizer is called before the program terminates[23].

In short, Java finalizers are not reliable. This is the other reason why they are not used. If you really need the functionality of a destructor in Java, the only thing you can do is to implement a public method which does the cleanup and call it explicitly.

Even [Blo01, Item 6] focuses on the topic of Java finalizers and gives the ultimate suggestion "Avoid finalizers".

3 finally blocks

Both languages know exceptions and support try/catch blocks to handle them. But only the Java programming language knows the so called finally blocks. Finally blocks follow a try block and can include code which will be executed regardless how the try block finishes. This means that the code in a finally block will be executed if the try block is left normally (no exception) as well as if there is a exception, and even if there are two (in the try-block and in a catch block). The finally block is always executed as last part of the try block, so in the presence of catch blocks, the catch blocks are executed first in case of exceptions [GJSB00, §14.19.2].

For that reason `finally` blocks are used to perform cleanup tasks. Listing 5 shows a commonly used Java structure[24]. In line 2 a resource

Listing 5: Java finally example

```
1 void JavaFinally() {
2   resource = new resource();
3   try {
4     // something which could
5     // throw an exception
6   } finally {
7     resource.cleanup();
8   }
9 }
```

is acquired (imagine), this resource needs to perform some cleanup tasks when it's no longer used.

To ensure that the cleanup will happen, regardless how the try block is left, the finally block will be used to call the `cleanup` method. A common application for this structure is a database connection pool. The acquire operation would, for example, get a database handle out of a pool and the cleanup would hand it back so that it can be reused . We will come back to this example in the next section.

[22] It's possible to tell the JVM that now is a good time to collect garbage, anyway there is still no guarantee that the unreferenced objects are deleted when the `System.gc()` call returns. The explicit call of the garbage collector is not required, it's just a hint to the JVM.

[23] C++ defines that static objects will be destructed at program termination [Str97, §10.4.3]. Since all objects on the stack will be destructed as well, only objects on the heap will not be destructed automatically. One limitation of this is that it does happen on abnormal termination like calling `_exit`, `abort` or similar functions.

[24] [AGH00, §8.4.1] does present another structure as well. We will use the shorter version for simplicity.

The acquire operation is placed outside the try block, so that if an exception happens in the acquire operation, the cleanup is not done.[25]

The C++ programming language does not have a direct equivalent of Java finally blocks. But as we have seen in section 1.2.2 there is something which, happens regardless how a block is left: Objects placed on the stack get deleted if the block in which they were declared ends. And if an object gets deleted, it's destructor is called (see section 2).

To benefit from this, we will use a technique mentioned earlier: Using smart pointer types.

The C++ language allows to override operators. There are only a few operator's which can not be overloaded, but the most operators used to work with pointers can[26].

The basic idea of a smart pointer is to implement a class which is compatible to a normal pointer and has a destructor which performs the cleanup. The STL defines the `auto_ptr<>`[27] class which implements such a pointer. This pointer implements the so called "destructive copy" technique. This means that there can be[28] only one `auto_ptr<>` which owns the object it points to. If this `auto_ptr<>` becomes deleted the destructor will perform a delete on the object it points to. This ensures that the destruction of the pointer will also delete the object it points to.

Listing 6 implements the same example as listing 2 using `auto_ptr<>`'s. The `#include` in line 1 makes the `auto_ptr<>` available[29].

The key difference is that the integer pointers (`int*`) have been replaced by `std::auto_ptr<int>` types[30], and the deletes have been removed.

Listing 6: C++ `auto_ptr<>` example

```
1  #include <memory>
2
3  void CPPauto_ptr() {
4    std::auto_ptr<int> x(new int(0));
5    std::auto_ptr<int> z;
6    {
7      *x = 3;
8      std::auto_ptr<int> hlp(new int(*x));
9      int& y = *hlp;
10     y++;
11     z = hlp;
12   }
13   (*x)++;
14   ++(*z);
15 }
```

Another difference to listing 2 is the handling of the reference `y`. The object where `y` references to is now also managed by a `auto_ptr<>` called `hlp`. To pass this object out of the inner block, we have to copy a pointer to a variable which is available in the outer block. This is done in line 11 similar to the previous example. By copying an `auto_ptr<>` like shown, there happens something magic: the ownership of the object is transfered to the left hand `auto_ptr<>`. The other one `hlp` looses its ownership, therefore it does not delete the object when it goes out of scope in line 12. Now the `z` variable is responsible for this object and will delete it if it goes out of scope (line 15).

Another problem mentioned in 1.2.3 is also solved by this technique: If, for example, an exception happens in line 8, the resources occupied by the previously allocated object (line 4) will be cleaned up properly. This is also the reason why each allocated object is directly passed to a `auto_ptr<>` (line 8). Using this technique guarantees to not leak if an exception happens.

A word of warning has to be placed about `auto_ptr<>`: it is not a generic pointer type, as you have seen it has some very uncommon behavior (transfer of ownership, destructive copy) which limits it's use. Please consult [Jos99, Chapter 4.2], [Str97, §14.4.2] and [Sut99, Item 37] about the details of `auto_ptr<>`[31].

Another pointer type which can be implemented the very same way is a garbage collected pointer type. Instead of implementing the ownership transfer as done with the `auto_ptr<>`, you just need to introduce an usage counter.

[25] To not loose any resources for cases where the acquire operation throws an exception, the acquire operation must be implemented *exception safe*. [Sut99] explains this term in detail.

[26] The pointer to members related operator `.*` can not be defined by a user (see [Str97, §11.2]). Anyway, this is far beyond the scope of this paper.

[27] The `<>` brackets indicate that this type is a template. [Jos99, Chapter 4.2] has a good introduction into `auto_ptr<>` for further reading.

[28] In fact, the developer has to take care that one object is owned by only one `auto_ptr<>`. The `auto_ptr<>` assists by transfer of ownership if it is copied, but it is still possible to construct two `auto_ptr<>`'s with the same object.

[29] Java and C++ have very different ways to handle libraries. There is no direct comparison to anything in Java.

[30] The leading `std::` specifies the namespace . The `<int>` specifies that this is an `auto_ptr` to an integer value.

[31] [Jos99] does include a standard compliant implementation of `auto_ptr<>` which has less than 100 lines of code.

This counter would need to be incremented if another pointer points to the same object and be decremented if one of the pointers referencing to this object gets deleted or gets a new object assigned. If the usage count drops to zero, it would delete the object it points to. The STL does not contain such a pointer, anyway it isn't magic, it just needs some practice to do it[32].

In fact, the C++ destructor can be used every time you would use a Java finally block. In contrast to a Java finally block, the solutions presented here allow better code reuse and make it impossible to forget a finally block. The next section will demonstrate a non memory related resource management using this technique.

4 final comparison

To have a comparison of both languages we will implement the solution to a common problem. For a database application you have a database pool which is able of handling connections to a database. The basic functionality is provided by the get() method which returns a database-handle that can be used to perform operations on the database. The release() method has to be used to hand the database-handle back to the pool to allow its re-use. For this comparison we will imagine that we have such a facility which we will just call `pool` for both languages. Furthermore we will use `handle` to name the database-handle class.

We will consider the Java code first. Listing 7 is a 1-by-1 implementation of the finally technique described in listing 5. In line 2 the database handle is acquired from the pool, the body of the try-block does the operations on the database and the finally block hands the resource back to the pool. Nothing surprising so far, thats a very common structure in Java.

Now lets try to implement this example with the C++ techniques described in the previous section. The first thing we need is "something similar to `auto_ptr<>` but for database connections". Listing 8 is a very short implementation of such a class[33]. Line 1 declares the class to be a template class which takes two type parameters R (the resource, handler in our case) and P (the pool). This is required to allow the class to be reused for other types of resources, anyway for this example you can imagine that `P` is replaced by `pool` and `R` by `handler` in the body of the gc_resource class.

The constructor in line 4 takes a pool as argument, stores it locally and performs the get() operation on it. The result (the handler, or a pointer to a handler in this example) is stored locally too[34]. The destructor in line 9 returns

Listing 7: Java database pool example

```
1  void DBPool(pool mypool) {
2    handle h = mypool.get();
3    try {
4      // something which could
5      // throw an exception
6      // e.g.: h.executeQuery()
7    } finally {
8      mypool.release(h);
9    }
10 }
```

Listing 8: C++ garbage collected resource

```
1  template<class R, class P>
2  class gc_resource {
3    public:
4      gc_resource(P a_pool):
5        _pool(a_pool),
6        _resource(_pool.get()) {
7      }
8
9      ~gc_resource() {
10       _pool.release(_resource);
11     }
12
13     R* operator->() {
14       return _resource;
15     }
16
17   private:
18     P& _pool;
19     R* _resource;
20 };
```

[32] There are a lot of implementations of such a garbage collected pointer. [Jos99, §6.8] presents such a implementation called `gc_ptr`. [Ale01, §7] implements a configurable smart pointer, this can be configured to use different ownership strategies, different testing, different thread safety and so on. Other then most smart pointer libraries, this one implements only one smart pointer which can be parameterized by template parameters. This allows many combinations of the parameters and makes it very easy to use exactly the type which fits to a requirement.

[33] The listing is only for educational purposes. Basically it will work well, but there are some aspects which should be considered before it is used for production code. One of the major points not covered with the shown implementation is the lack of handling copy assignment and copy construction.

[34] The syntax used might look ugly, thats the syntax for C++ member object initialization. In that example it would also work if you would place the described logic in the body of the constructor. In this case, the only difference is a very minor performance drawback. But

the resource to the pool using the release() operation of the pool where the resource came from.

The function in line 13 overloads the -> dereference operator, this is required to access the handler from outside the gc_resource class. In fact it would be possible to implement a normal public method which returns the pointer (in Java you have not other choice), by overloading the dereference operator it is possible to use this class as you would use a plain pointer.

Listing 9 does implement the same exception safe code as listing 7 does for Java. As you notice, there is no try-block. There is not even code that releases the database handle in the end. This is all handled by the destructor of the gc_resource variable `h` which is called if the block ends in line 7.

Listing 9: C++ database pool example

```
1 void DBPool(pool mypool) {
2   gc_resource<handle, pool> h(mypool);
3
4   // something which could
5   // throw an exception
6   // e.g.: h->executeQuery()
7 }
```

Of course you can still add a try/catch block to handle possible exceptions, but for the management of the handler isn't required anymore.

On the first look, both implementations are equivalent. The benefit of the C++ implementation is not to save some lines each time you have such a structure (by the cost of some lines for the implementation of gc_resource). The benefit is that there is nothing which can be forgotten. By acquiring the resource the required cleanup code is automatically installed.

On the other side, the Java implementation doesn't need a special class for this, and most importantly it doesn't need the know how to implement this class. For sure the Java implementation is more straight, and easier to understand without preparation. But of course it has also drawbacks, for example that the programmer has to do the right thing every time again, this does not only introduce duplicate code it is error prone. For example somebody could forget to release the handler, or do it at a wrong place like a catch block or even without any exception handling.

So whats better? It depends. In that case, the best technology depends on the size of the project. For smaller projects the Java solution will be fully sufficient and just more straight. For big projects the C++ solution might become more interesting, since it allows the elimination of duplicate code (which is error prone) by the introduction of a small class like presented in listing 8.

there can be cases where it makes a functional difference: if an exception occurs

References

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition, 2000.

[Ale01] Andrei Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

[Blo01] Joshua Bloch. *Effective Java*. Addison Wesley, 2001.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library*. Addison Wesley, 1999.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Sut99] Herb Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison Wesley, 1999.