International Journal of Current Research and Applied Studies, (IJCRAS)

ISSN: 2583-6781

available at https://ijcras.com/

Volume 3 Issue 4 July-Aug 2024

Page 01-11

MEMORY MANAGEMENT SYSTEM AS APPLIED BY C++, JAVA, C# AND ASSEMBLY LANGUAGES IN THE OPERATING SYSTEM

Uyoo, Stephen Yavenga[1] Francis, Akogwu Alu[2] and Rimdans, Victor Zwalmak[3]

[1]Computer Science Department, Joseph Sarwuan Tarka University, Makurdi, Nigeria
[2]Information Security Analyst, USA
[3]National Mathematical Center Abuja, Nigeria

DOI: https://doi.org/10.61646/IJCRAS.vol.3.issue4.82

ABSTRACT

In the recent era of computing, applications and operating systems cannot survive without efficient memory management, especially if an application has to be under survey load for an undefined long time. Efficient utilization of resources will enhance performance. This paper describes the memory management in an operating system and it will demonstrate basic architecture of segmentation in an operating system and basic of its allocation. This paper also describes the basic concept of virtual and dynamic memory management and also review the languages used in system programming such as, C++, java, .NET and assembly languages exploring their issues and techniques in memory management.

Keywords: Smart Pointer, Activation Record, Garbage Collector, Call stack, Allocation and Recycling

INTRODUCTION

Memory management is the procedure of governing and organizing computer memory, allocating portions called blocks to various running programs to enhance overall system performance. Memory management resides in hardware, in the Operating System, and in programs and applications (Kabari, L. 2015).

C++ and Java both have memory management techniques. This paper will analyze the differences between their memory management approach, garbage collectors and its issues. Basically, C++ does not have

efficient garbage collector but it has some means to achieve this through smart pointer and RAII (Resource Acquisition Is Initialization); *a programming expression used in numerous object oriented, statically typed programming languages to define a specific language performance*. To define the memory function, we should understand the call stack, which is very important for every programming language. Every function main job is to return value to the memory about its status. Therefore, call stack has to be maintained throughout the program and explains how call stack works in C++ and Java. Heap memory is another powerful memory in the system which comes to play when the program needs memory as a block to be allocated from the heap memory by operating system to efficiently prevent deadlock conditions. Dynamic memory allocation from heap memory is used by C++ and Java. The smart pointer and garbage collector plays a major role when analysing how heap memory is used by C++ and Java. (Eltaeib *et al.*, 2015).

Memory management is elementary for all languages because it is the most significant aspect to determine the effectiveness of the language. Languages like Java have their own garbage collector hence the programmer does not need to do memory management. The programmer uses a **new** and **delete** functions in C++ to release the memory. Here we will discuss how the memory is managed in both languages and issues in the memory management of the languages.

Memory management contains components that physically store data in hardware, such as flash-based SSDs (solid-state drives), RAM (random access memory) chips, and memory caches. Memory management in the Operating System comprises the allocation (and constant reallocation) of specific memory blocks to single programs as user demands alteration. At the application level, memory management certifies the availability of sufficient memory for the objects and data structures of each running program continuously. Application memory management syndicates two related tasks, known as recycling and allocation (Rahman, M.M. 2022).

- Recycling: As soon as a program no longer needs the data in previously allocated memory blocks, those blocks become obtainable for reassignment. The programmer manually handles this task or automatically by the memory manager.
- Allocation: Once a block of memory is requested by the program, a part of the memory manager called the allocator allocates that block to the program.

2. MEMORY MANAGEMENT IN C++

Vassev & Paquet. (2006). Memory management is undoubtedly the prevalent issue when using C++. Memory management inaccuracies are both easy to create and difficult to detect. C++ implies three built-in memory management problems:

- ✓ It is possible to allocate memory and never de-allocate it, even after all pointers to it have been destroyed. This results in memory leak.
- ✓ C++ does not check array bounds. Hence, problems occur when insufficient memory is allocated.

- ✓ Pointer management is another issue that arises when pointers denote objects. Improper use of pointers results in *dangling references*, which occur when memory is allocated and then de-allocated but the pointer is still in use.

2.1 Memory Management Strategies.

Three kinds of memory management In C++ are recognized:

i. Automatic (Stack Allocation)
ii. Static Allocation and
iii. Dynamic Allocation.

Automatic management (stack allocation) is restricted to dealing with local variables. In this case, the memory allocated for a local variable is de-allocated automatically at the end of the variable's scope. Hence, it is dangerous to use pointers with local variables, since this is a potential dangling reference case (Choi et al., 2003).

Static allocation is used for storing global variables and variables declared static (both local static variables and static class members). Those variables have to persist for the entire run of the program. In static allocation, a certain range of memory is set aside for the storage of static variables, and this memory can never be used for anything else for the duration of the program.

Dynamic allocation (heap allocation) is requested by using the *new* and *delete* keywords. As a rule, C++ does not automatically de-allocate anything allocated with new, i.e. we must use the *delete* keyword to free dynamically allocated objects. We should mention here that *new* and *delete* are potentially expensive operations, "because of the extra bookkeeping the memory manager must do, and often because of the extra bookkeeping the programmer must do". (Martin S. 2019).

In addition, C++ implements a function *malloc()* that like new dynamically allocates memory. The *malloc()* function returns a pointer of type void to a memory buffer of the requested memory block.

2.2 Allocating and De-allocating Objects and Arrays of Objects.

Consider the following definition:

Asteroid* astr = new Asteroid() ;

Here the expression *new Asteroid()* allocates an object of type Asteroid. The object is initialized by default (the constructor by default has been called) and a pointer to this newly allocated object is returned as a result. The memory allocated for this object will be de-allocated when the delete is executed on the object's pointer as following:

delete astr;

In case we want to allocate memory for an array of asteroids, we call new steroids[n] that allocates an

array of n objects of type asteroid and returns a pointer to the initial element of the array.
Asteroid* astr = new Asteroid[n];

In order to de-allocate the memory allocated for this array we call:
delete[] astr;

Here the brackets tell the system to de-allocate the memory for the entire array. By executing new and delete on an object's pointer, we actually call the object's constructor and destructor respectively.

Koenig and Moom. (2000), put the enunciation on the fact that C++ allows allocating an array with no elements - new Asteroid[0]. They examine this fascinating performance and settle those new returns in that case not a pointer to the first element but "a valid off-the-end pointer". This pointer points to the address where the initial element shall be if there is such. Following that logic, Koenig and Moo recommend a unique use of an array in union with an STL container:
T* p =new T(n);
Vector<T> v(p, p+n);

2.3 Solutions Generally Applied to Memory Management in C++
Prototypes help a lot in finding solutions to the memory management problems in C++. By depending on models, we can create semi-automatic solutions to the memory management problems in C++.

Smart Pointers. These are *reference counting pointer objects* suitable for de-allocating memory. *Reference counting* is a semi-automated memory management technique, meaning that it obliges some programmer support, but it does not entail one to know when an object is no longer in use, i.e. the reference counting machinery is accountable for that. (Jonathan B. 2004).

A smart pointer class overloads certain operators like *operator*, operator->* and others. This gives an impression for the smart pointer object of being a real pointer.

Template handle class. (Koenig and Moom. 2000), exhibit a generic prototype class called *handle* that captures a handle behavior, i.e. it refers to an object and when we destroy the handle object, it will destroy the associated object as well.

Auto Pointer. (Grogono, 2005). Presented about an effective and efficient smart pointer called "*Auto pointer (auto_ptr)". The auto_ptr* holds a dynamically allocated object and "achieves automatic cleaning once the object is no longer needed". (C/C++ Users Journal, 1999). An *auto_ptr* is a pointer that cloaks another object pointer and automatically de-allocates the object pointed by the cloaked pointer once the *auto_ptr* goes out of range. This procedure is potential because the *auto_ptr* is simply used as an automatic object – one that is destroyed automatically when it goes out of scope. Hence, the basic idea with auto pointers is that they should have an owner that "deletes the object pointed to". Some complications

revealed with auto pointers.

1. The incompatibility of auto pointers with the STL containers and the inefficiency of auto pointers in multithreaded environments.
2. Another problem with auto pointers is that "for auto_ptr, copies are not comparable". *(C/C++ Users Journal, 1999).*

Const auto pointer. A const auto pointer is an auto pointer that never loses ownership, because it is declared as a const and never changes.

Counted pointer. (Colvin, 1994) recommended a modified **auto_ptr** termed **counted_ptr.** This smart pointer transforms the auto_ptr by making available a working copy constructor and assignment operator.

Given the analysis above, it is determined that memory management in C++ is difficult and a complex task, but also there are many general solutions that systematize the memory management process. The major issue is that none of them works competently in all the aspects of C++ programming. However, in some aspects of computer programming the flexibility and power to control the overall memory management process is an immense advantage.

3. MEMORY MANAGEMENT IN JAVA

One of the reasons why Java is so prevalent is that it spontaneously manages all the memory allocations and de-allocations. Considering the development time this is an immense advantage.

Memory allocation and de-allocation. In Java every variable is an object, except those of primitive types like *boolean, char, int* etc, Java allocates memory only to objects, (Java Q&A Experts, 2005). i.e. in contrast to C++ there is no *malloc*() function. Innovative objects are produced by the innovative operator in Java in virtually the same way as in C++. This allocates memory for the object and also calls one of the constructors (David G. M. 2018).

Vassev & Paquet (2006). In contrast to C++, Java does not do with the *delete* operator. In addition, the Java language requirement does not allow definition of a class destructor. As an alternative, the Java scheme automatically de-allocates objects once no references to them remain. This is done by a process within the Java Virtual Machine (JVM) called *garbage collector*. In addition, a java object has a technique called *finalize*(), which technique is called once the object is de-allocated. This is coarsely an analog to the C++ *destructor*. This routine can be overridden to perform some tidying up tasks when an object is *"garbage collected*". However, there is no way to predict when the garbage collector will de-allocate an object or whether it will do it at all for any specific entity.

Java pointers. It is occasionally denied that Java does not have pointers. In fact, apart from the variables of the original types, every Java variable is a pointer. For the reason that this restraint, Java has no special

notation for declaring a pointer, nor is there any notation for computing an address or dereferencing a pointer. Given the investigation above, it is concluded that Java programming language comparing to C++ is much modest. Hence, the implementation of a facility like *garbage collector* is much easier than in C++.

Garbage collection is an approach for instinctively detecting memory allocated to objects that are no longer functional in a program, and recurring that allocated memory to a pool of unrestricted memory locations. This technique is in distinction to "manual" memory management where a programmer plainly codes memory requirements and memory releases in the program. Automatic garbage collection has the benefits of reducing programmer load and averting certain kinds of memory allocation bugs, garbage collection does require memory resources of its own, and can contest with the application program for processor time (David G. M. 2018).

Garbage collection is a Java's internal mechanism for automatic memory management. The last provides a full automatic detection and removal of data objects that are no longer in use. A philosophical analysis of Java garbage collection mechanism is done and specifies the basic approach for such a mechanism as following: References to objects are counted by starting "from every object that is statically accessible and follow all of their references" (Gillam, 2018).

Java implements a garbage collection machinery called *tracing garbage collection*, "because it touches all the possible reference paths". We observe two major categories of garbage collection and many hybrid versions: (Gillam, 2018).
- *Mark-sweep garbage collection,*
- *Copying garbage collection,*

Hybrid approaches
- *Mark-compact garbage collection,*
- *Generational garbage collection, and*
- *Incremental garbage collection*.

Garbage Collection Process Overheads.
Gillam, R. (2018). The garbage collector as a program unit requires memory and CPU time. In addition, for the search operations, it uses the machine's stack, "raising the danger of stack overflow".

Finalization. Previously seen above, in distinction to C++, Java does not implement class' destructor, but does implement a finalizing function. Since, there is no need to explicitly destroy the objects, why we need finalization. The answer is to avoid potential *dangling references* coming from operating-system data structures, files, counts or other internal statistics, and so on.

Disadvantages of Garbage Collection: (Gillam, 2018).

- ✓ In C++, smart pointers can be used to allow taking ownership in a constructor and emancipating ownership in the destructor. When a function uses a smart pointer, the function allocates the applicable wrapper object (*auto_ptr* object) on the stack, and the resource is spontaneously unrestricted when the wrapper object goes out of scope. This idiom does not work with garbage collection.
- ✓ In C++, an object can do a full clean up, by discharging everything not just subordinate data structures. In Java, the garbage collector manages only memory.
- ✓ Garbage collection has some problems with copying and sharing objects.

Assumed the analysis above, it is determined that there is definite amount of tradeoffs involved in using a garbage-collected language such as Java versus a language such as C++ that uses manual memory management. As we saw these tradeoffs are related to:

- ✓ Some language restrictions that do not allow implementation of such idioms like *auto_ptr*;
- ✓ The need to clean up non-memory resources manually, since garbage collection does not deal with them.
- ✓ The collection process overhead, which sometimes could result in stack overflow;
- ✓ Copying and sharing objects;

4. MEMORY MANAGEMENT in C#

Memory management in C# is spontaneous, which frees developers from manually allocating and releasing the memory occupied by objects. Garbage collector implements Automatic memory management policies. (Huynh & Roychoudhury, 2006). The object memory management life cycle of is as follows.

1. Memory is assigned for every object created, the constructor is run, and the object is reflected live.
2. Once the destructor for an object is run, if that object (or any part of it) cannot be accessed by any possible continuation of execution, including the running of destructors, the object becomes inaccessible and considered eligible for collection.
3. Once the object is eligible for destruction, at some unspecified later time, the destructor (§10.12) (if any) for the object is run. Except succeeded by explicit calls, the destructor for the object is run once only.
4. Once one portion of the object cannot be read by any possible continuation of execution, other than the running of destructors, the object is reflected no longer in use, and it becomes fit for destruction. The C# compiler and the garbage collector may choose to analyze code to ascertain which references to an object may be used in the future. For example, if a local variable that is in scope is the only existing reference to an object, but that local variable is certainly not referred to in any imaginable continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.
5. Finally, at some point after the object becomes suitable for collection, the garbage collector frees the memory supplementary with that object.

The garbage collector retains information about object procedure and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or isolated. Other languages that accept the presence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not entail that destructors be run or that objects be collected as soon as they are eligible or that destructors be run in any precise order or on any specific thread. The performance of the garbage collector can be controlled, to some degree, via static methods on the class System. This class can be used to demand a collection to occur, destructors to be run (or not run), and so forth. (Huynh & Roychoudhury, 2006).

For the reason that the garbage collector is permitted wide latitude in determining at what time to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program creates an instance of class A and an instance of class B.

5. ASSEMBLY LANGUAGE

Assembly Language is a type of low-level programming language that is projected to interconnect straight with a computer's hardware. Contrasting machine language, which encompasses of binary and hexadecimal characters, assembly languages are aimed to be readable by humans. **Assembly language** is more than low level and less than high-level language so it is an intermediary language. Assembly languages use numbers, symbols, and abbreviations in places of 0s and 1s. For example: For multiplications, addition, and subtraction it uses symbols likes Mul, Add, and sub etc. (Encyclopaedia Britannica, Inc. 2021)

Assembly language is used principally for hardware operation, access to dedicated processor instructions, or to address critical performance issues. Distinctive uses are device drivers, low-level embedded systems, and real-time systems. They are frequently used to write operating systems, so they are sometimes called system programming languages. Programs written in mid-level languages can achieve as well, or nearly as well, as programs written in assembly language. Examples of mid-level programming languages include Nim, C++**,** Rust, C, and java.

Machine language is the **low-level programming language**. It can only be denoted by 0s and 1s. Assembly languages use numbers, abbreviations, and symbols instead of 0s and 1s. Aforementioned when we want to generate a picture or show data on the screen of the computer then it is very challenging to draw by means of only binary digits (0s and 1s). For example: To write 120 in the computer system its depiction is 1111000. So, it is very tough to learn. To overcome this problem the assembly language is designed. (Encyclopaedia Britannica, Inc. 2021)

Assembly Language Memory Management (**Memory Segments)**

A segmented memory model splits the system memory into clusters of autonomous segments, referenced by pointers located in the segment registers. Each segment is used to hold a precise type of data. One segment is used to hold instruction codes, another segment keeps the data elements, and a third segment keeps the program stack. In the light of the above discussion, we can stipulate various memory segments

as:

- ✓ **Data segment** - This is characterized by data segment and the block starting symbol (bss). The data segment is used to declare the memory region where data elements are kept for the program. This segment cannot be lengthened after the data elements are declared, and it remains stationary all through the program. The bss segment is also a stationary memory segment that holds buffers for data to be declared later in the program. This buffer memory is zero-filled.
- ✓ **Code segment** - This is denoted by text section. It describes an area in memory that keeps the instruction codes. It is a fixed area.
- ✓ **Stack** - this section contains data values processed to functions and procedures within the program.

6. CONCLUSION

In sections 2 and 2.1, we saw that the memory management exposed by C++ is powerful and sufficient, since it gives a full control of the memory management process, but also the lack of automatic memory mechanism in C++ often results in dangling pointers and memory leaks. There are many generic solutions, like smart pointers, that help in improving the C++ memory management. The general idea in those solutions is that we could automate the memory management process by using reference counting, but this automation is semi-automation and it does not work for all the aspects of C++ programming. The conclusion is that C++ definitely needs a mechanism for automatic memory management, with preserving the current powerful manual memory management mechanism.

As we saw in section 3, Java with its garbage collection mechanism ensures easiness and efficiency in programming. The garbage collection prevents errors like memory leaks and dangling references (at least for the memory resource). Hence, the programs are less error-prone. Unfortunately, not everything about garbage collection is perfect. There are numerous reasons why a program in Java might be three to nine times slower than the same in C++. One of the reasons with no doubts is the garbage collection. As we saw in section 3, the overhead in the garbage collection process is one of the many tradeoffs, which we have with the Java garbage collection mechanism.

Given the reasoning above, we conclude that garbage collection is a good advantage for Java, since Java is simpler and less powerful language than C++ and the tradeoffs coming with the garbage collection do not have measurable impact on the Java language. However, implementing such facility in C++ is far more complex, since it will drastically reduce the power of the language. Therefore, both kind of memory management are appropriate for their usage, but the C++ model still needs to be improved by adding to the current model autonomic garbage collection mechanism.

C and C++, java and assembly language are some of the languages used to describe memory management of an operating system and also a short outlook of memory management. In this paper, different memory allocation techniques are discussed along with their comparative analysis.

When compare the two languages such as C++ and JAVA, both have some sort of advantages and disadvantages. C++ is said to have more memory leaks it can be controlled by some point with smart pointer and RAII. These techniques are implemented manually or it is semi-automated but there should be a technique which is fully automated like Java. As we see in Java garbage collector, it is fully automated but sometimes like in multi-threading it creates "Dangling pointer" reference. The garbage collector has to search and collect the memory from heap it will slow down the process and that hence Java is not as powerful like C++ which is faster than Java. So, the technique has to improve in java as in performance level and some of the research like "RTJ (Real Time Java)" has try to solve this real time problems. Still in C++ automation is needed for memory management which should not compromise language efficiency.

REFERENCES

*David G. M., (2018) Java: A System Programming Language. Supplementary section for Understanding Networked Applications: A First Course, Morgan Kaufmann.

*Kabari, Ledisi. (2015). Efficiency of Memory Allocation Algorithms Using Mathematical Model. International Journal of Emerging Engineering Research and Techn ology Volume 3, Issue 9, September, 2015, PP 55 - 6 7 ISSN 2349 - 4395 (Print) & ISSN 2349 - 4409 (Online).

*Rahman, Mohammad Mushfequr (2022) Memory Management in Computer System.

*Vassev, Emil & Paquet, Joey. (2006) Aspects of Memory Management in Java and C++. 952-958. https://www.researchgate.net/publication/221610499_Aspects_of_Memory_Management_in_Java_and_C/citation/download. Retrieved; 20th, Aug. 2023.

*Koenig, A., Moom, B. E. (2000) "Accelerated C++", Pearson Education Inc., Boston.

*Jonathan Bartlett, (2004) "Inside memory management", IBM, New Media Worx. http://www-128.ibm.com/developerworks/linux/library/l-memory/

*Peter Grogono, (2005) "Software Development with C++", Lecture Notes, Concordia University, Quebec, Canada, 2005

*C/C++ Users Journal (1999) "Using auto_ptr Effectively", C/C++ Users Journal, 17(10), October 1999, http://www.gotw.ca/publications/using_auto_ptr_effectively.htm. Accessed; 21st Aug. 2023.

*Gregory Colvin, (1994) "Exception Safe Smart Pointers", C++ committee document 94-168/N0555. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1994/N0555.pdf. Accessed; 21st Aug. 2023.

*Richard Gillam, (2018)" An Introduction to Garbage Collection", IBM Center for Java Technology-Silicon Valley, http://www.concentric.net/~rtgillam/pubs/Garbage1.html. Accessed; 20th August, 2023.

*Huynh, T. & Roychoudhury, A. (2006) A Memory Model Sensitive Checker for C#. 476-491. 10.1007/11813040_32. Prashant Shenoy, 2021. Lecture notes, CMPSCI Operating System.

*Eltaeib, Tarik & Sengottaiyan, Gayathri. (2015). MEMORY MANAGEMENT IN C++ AND JAVA. International Journal of Engineering and Computer Science. 4.

*Martin Sperens (2019) "Dynamic Memory Managemental C++". Computer Game Programming, bachelor's level. Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering. http://www.divaportal.org/smash/get/diva2:1367692/FULLTEXT01.pdf.

Retrieved; 20th Aug. 2023.

*Choi, Jong-Deok & Gupta, Manish & Serrano, Mauricio & Sreedhar, Vugranam & Midkiff, Samuel. (2003). Stack allocation and synchronization optimizations for Java using escape analysis. ACM Trans. Program. Lang. Syst.. 25. 876-910. 10.1145/945885.945892.

*Encyclopaedia Britannica, Inc. (2021) Computer programming language. Encyclopaedia Britannica. https://www.britannica.com/technology/computer-programming-language. Access: October 18, 2021