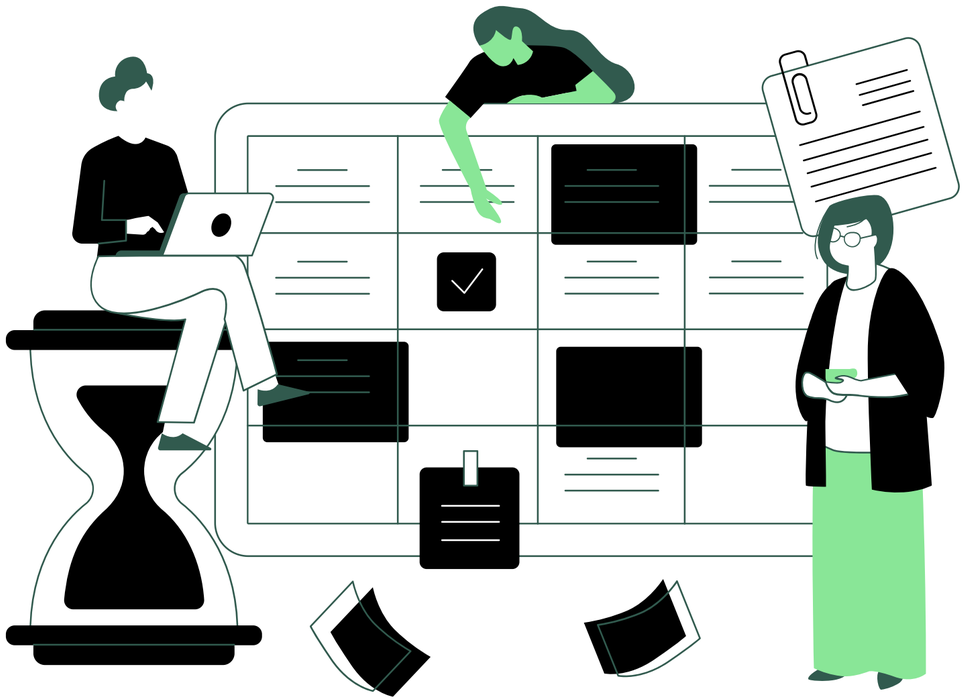
MEMORY MANAGEMENT & GARBAGE COLLECTION

Prepared by
YUVRAJ SINGH

Matters on the Docket

A brief look at what we will discuss on this topic



01 Memory Management and Its Types

02 Manual Memory Mangement

03 Garbage Memory Management

04 Types of Garbage Collection

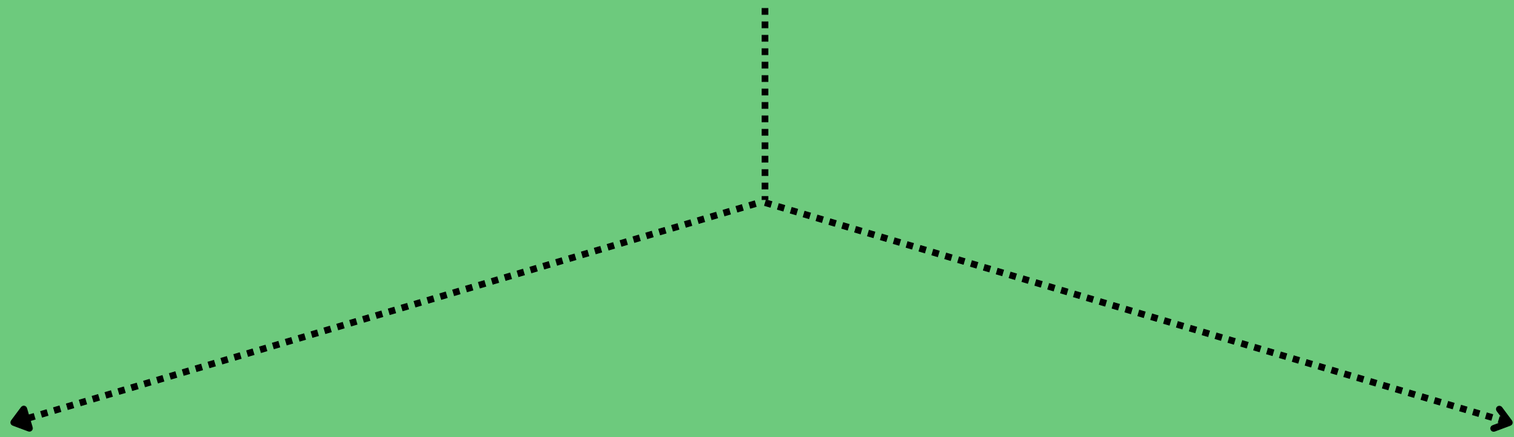
05 Ownership model in Rust

What is MEMORY and why MANAGE it !

The term Memory can be defined as a collection of data in a specific format. It is used to store instructions and process data.

Computer memory is a finite resource. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

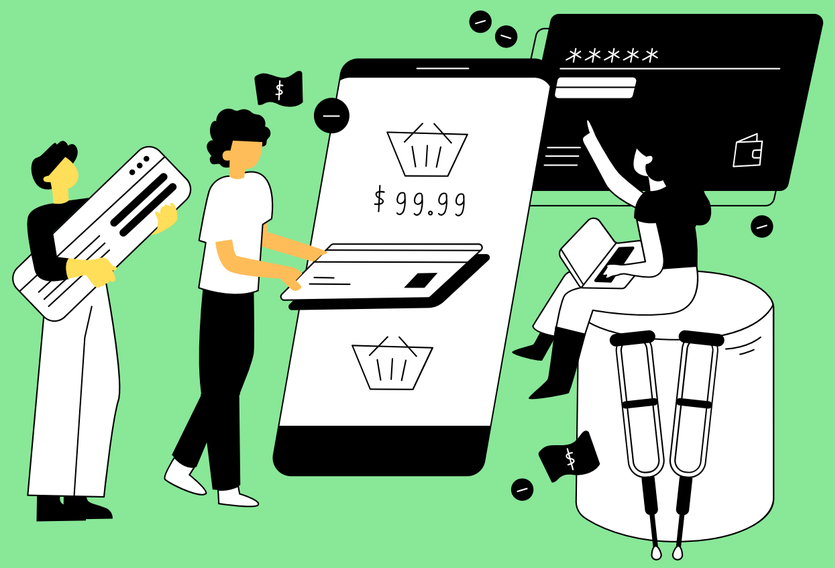
TYPES OF MEMORY MANAGEMENT



MANUAL MEMORY MANAGEMENT

GARBAGE MEMORY ALLOCATION

MANUAL MEMORY MANAGEMENT



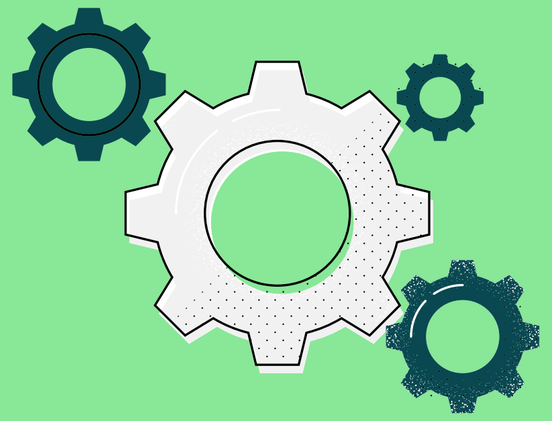
As the name suggests “*MANUAL*” means it requires programmers to manually free or deletes the dynamically allocated memory.

It mainly focuses on the Programmer and places all the responsibilities on them for memory management .

It requires strict programming discipline to ensure the long-term correctness and reliability of the programs.

The programming languages that use this type of memory allocation are mainly C and C++.

Pros AND Cons



PROS

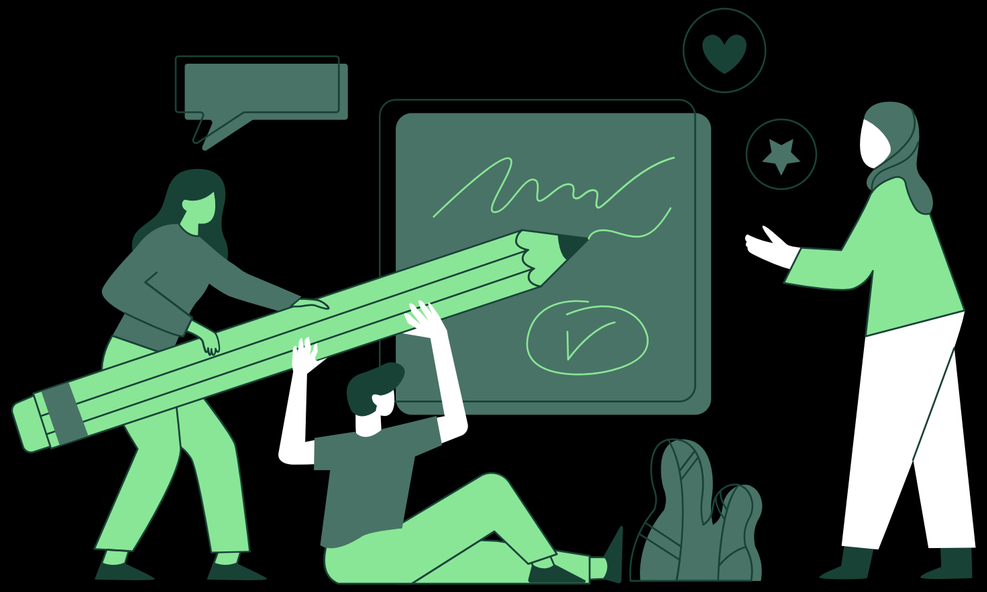
The biggest advantage of using this method of memory allocation is that it provides the fastest mechanism if all the things are done right.

CONS

Some common coding pitfalls include:

- **Dangling pointers due to releasing a live object**
- **Memory corruption due to double free**
- **Memory leaks due to forgetting to release memory**
- **Higher coupling between components due to modeling of ownership semantics in API signatures**

GARBAGE MEMORY ALLOCATION

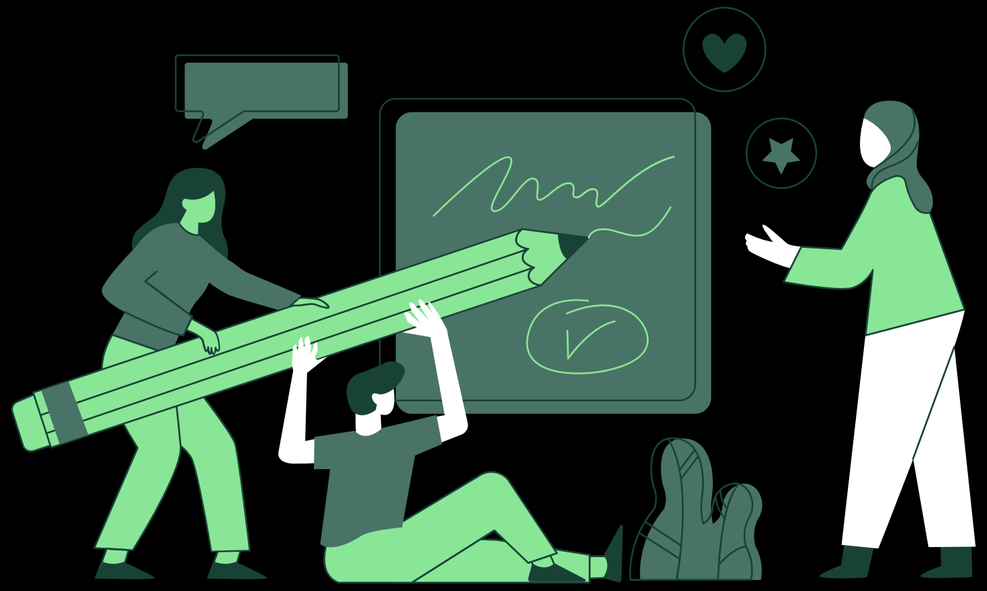


Garbage is the memory that was once used by objects but will never be read or written by the program again.

Garbage collection was invented by American computer scientist John McCarthy around 1959 to simplify manual memory management in Lisp.

This method releases the memory when not in use automatically which helps the programmer from managing the memory.

GARBAGE MEMORY ALLOCATION



Garbage collector is the magic component that makes the illusion of infinite memory possible. It attempts to reclaim garbage or memory occupied by objects that are no longer in use by the program.

This is the automated system for memory allocation which is highly used in famous programming languages like JAVA, python, .NET, etc

Advantages

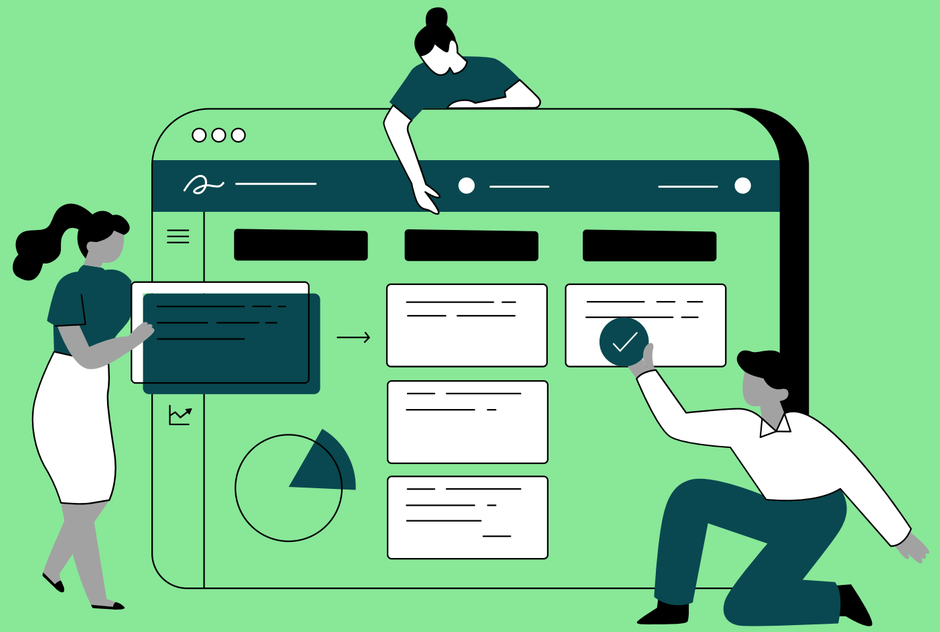
Garbage collection frees the programmer from manually dealing with ***memory deallocation.*** As a result, certain categories of bugs are eliminated or substantially reduces:

- ***Dangling pointer bugs***: occurs when a piece of memory is freed while there are still pointers to it and one of those pointers is dereferenced.

- ***Double free bugs:*** occurs when a program tries to free a region of memory that has already been freed.

- ***Memory leaks:*** when a program fails to free memory occupied by objects that have become unbreachable and can lead to memory exhaustion.

Disadvantages



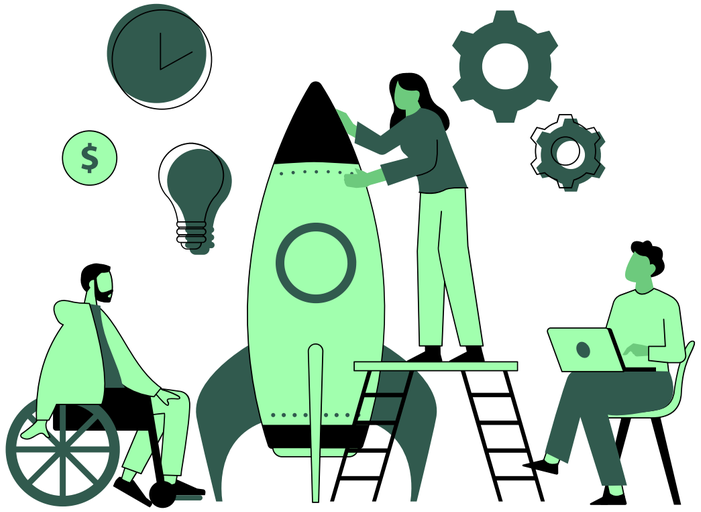
Garbage collection has certain disadvantages including consuming additional resources, performance impacts, and possible stalls in program execution.

Garbage collection ***consumes*** computing resources in deciding which memory to free.
The moment when garbage is actually collected can be unpredictable resulting in ***stalls***(pause shift/free memory)scattered throughout a session.

Unpredictable stalls can be ***unacceptable*** in real-time environment transaction processing, or in interactive programs. Incremental, concurrent, and real-time garbage collectors address these problems, with varying trade-offs.

Types of Garbage Collection

TRACING

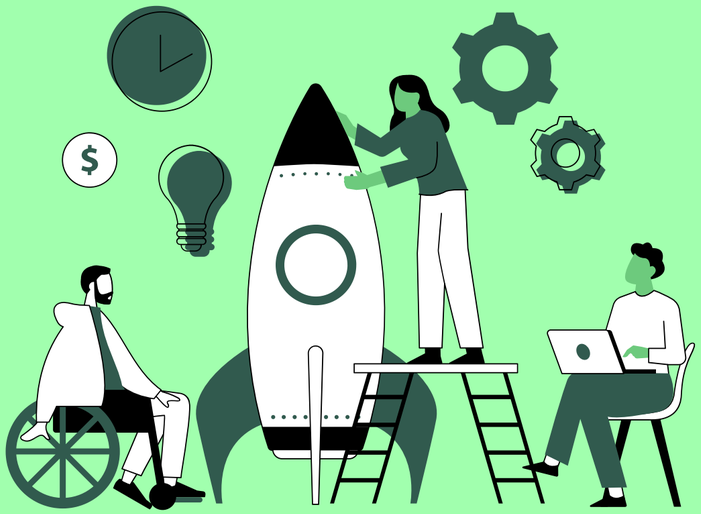


Tracing garbage collection is the most common type of garbage collection.

The overall strategy consists of determining which objects should be garbage collected by tracing which objects are reachable by a chain of references from certain root objects, and considering the rest as garbage and collecting them.

However, there are a large number of algorithms used in implementation, with widely varying complexity and performance characteristics. The languages that use this type are JAVA and .NET

Naive Mark-and-Sweep

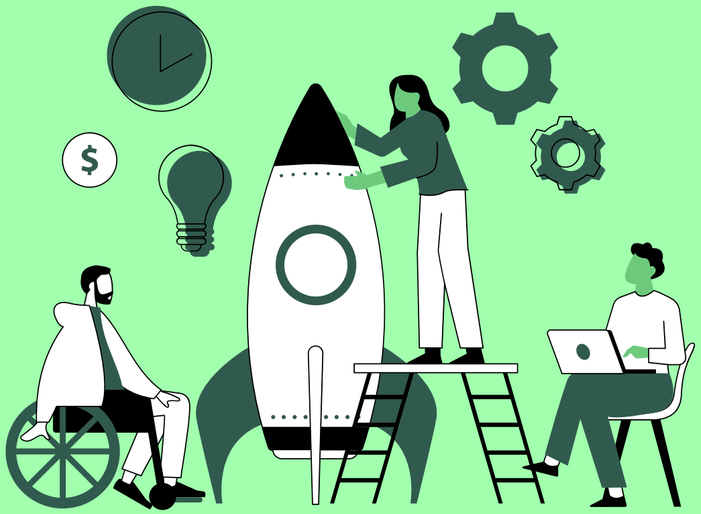


lIn the naive mark-and-sweep method, each object in memory has a flag (typically a single bit) reserved for garbage collection use only. This flag is always cleared, except during the collection cycle.

The first stage is the mark stage which does a tree traversal of the entire ' root set ' and marks each object that is pointed to by a root as being 'in use'.

All objects that those objects point to, and so on, are marked as well so that every object that is reachable via the root set is marked.

Naive Mark-and-Sweep



In the second stage, the sweep stage, all memory is scanned from start to finish, examining all free or used blocks; those not marked as being "in use" are not reachable by any roots, and their memory is freed. For objects which were marked in-use, the in-use flag is cleared, preparing for the next cycle.

This method has several disadvantages, the most notable being that the entire system must be suspended during collection, and no mutation of the working set can be allowed.

This can cause programs to ' freeze ' periodically (and generally unpredictably), making some real-time and time-critical applications impossible.

Reference Counting

Reference counting is a programming technique for storing the number of references, pointers, or handles to a resource, such as an object, a block of memory, disk space, and others.

In garbage collection algorithms, reference counts may be used to deallocate objects that are no longer needed.

Objects are reclaimed as soon as they can no longer be referenced and in an incremental fashion, without long pauses for collection cycles and with a clearly a defined lifetime of every object.

Reference Counting

In real-time applications or systems with limited memory, this is important to maintain responsiveness. Reference counting is also among the simplest forms of memory management to implement.

The frequent updates it involves are a source of inefficiency.

Reference counting requires every memory-managed object to reserve space for a reference count.

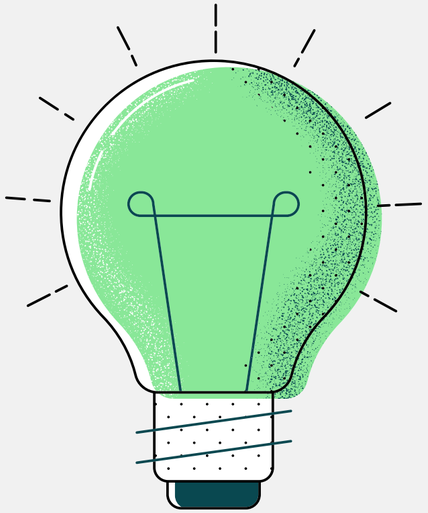
The naive algorithm described above can't handle reference cycles, an object which refers directly or indirectly to itself. Languages such as Python, JavaScript, and java use these methods.

Ownership Model of RUST

Ownership is Rust’s most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector.

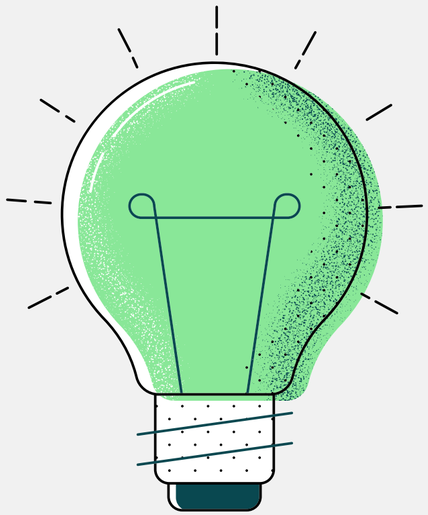
Rust stores Data in two different structure parts of memory :

- Stack
- Heap



Stack is a data structure that follows the principle of LIFO(Last In, First Out) and it is the place where rust stores data with a known, fixed size. For example, if an integer(32bits) has been assigned a value then both the variable and values will be stored in the stack. But if we take a mutable data type as a vector or String then it changes size,so when the program is running then the object is stored in the stack with a pointer to the heap, which stores the value of such data type.

Rules of Ownership

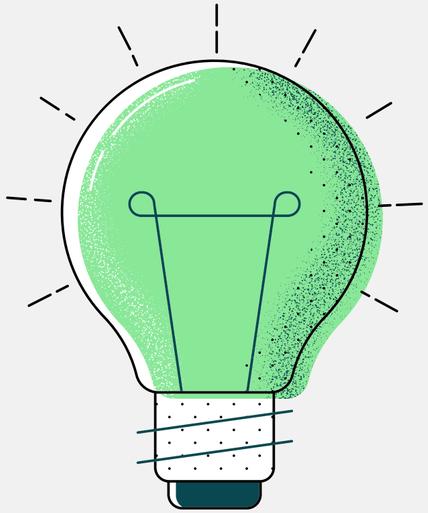


To prevent memory safety issues like Dangling pointers, Double free and memory leaks and to provide a fast mechanism Rust has made the ownership model which has following set of rules:

- Each value in Rust has a variable that is called its *owner*.
- There can be only one owner of a value at a time.
- When the owner goes out of scope, the value will be dropped.

This means that when we pass a variable to a new function or another variable, we transfer the ownership to that function or variable and we have no longer access to that variable. But we can access that variable with the concept of Borrowing.

Borrowing



To use a variable without invalidating its ownership we can either use .clone() method for creating a clone or simply use the concept of "Borrowing", which is we create a reference to a value by using the "&" sign,so it will point to the reference of variable rather than the variable itself and hence saving the ownership to the variable.

Rules for Borrowing:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must be valid.

THANK YOU