Why garbage collector is not necessary

Nowadays the modern programming languages need to provide tools to allow and make easy memory management, because memory is a non-infinite resource. There are two main philosophies affecting memory management:

- The first is based on the point of view that "*Memory management is too important to be left to the system"*: it must be managed by the programmer, who has available some means to allocate and deallocate memory (like functions malloc(), calloc(), realloc() and free() in C). However, because the developer is responsible of the deallocation of memory, it is easy to make mistakes, which could crash the whole system due to memory leak. The usefulness is that he has the complete control of memory management, and he decides when and how much memory must be deallocated.
- The second argues, instead, that "*Memory management is too important to be left to the programmer":* the system provides a service for the allocation/deallocation that is absolutely viewless to him, i.e. garbage collection. Garbage collection, or automatic memory management, provides significant software engineering profits over explicit memory management. It frees developer from the burden of memory management, eliminates most memory leaks, and improves modularity, while preventing accidental memory overwrites ("dangling pointers").Because of these advantages, garbage collection has been incorporated as a feature of a number of mainstream programming languages, like Java, Python and C# (.NET). Especially in Java objects are automatically allocated and deallocated when system deems it necessary and appropriate, as it keeps track of active references of each object; so garbage collector will release memory occupied by objects when it will be no more accessible**.** However, the programmer loses the control of memory management; furthermore, there is a moderate performance degradation (for example, at fixed intervals, the JVM runs the garbage collection thread, and this may affect programs that have specific targets and tight time constraints).

What is exactly a garbage collector?

We speak about automatic reclamation of computer storage or automatic management of dinamically allocated storage: more easily, garbage collector is based on search and destroy algorithms; they aren't looking for things to delete, they know exactly what has to be removed. It is to be consedered as a special routine, a real low-priority thread that works concurrenty to main program, unmanaged by programmer, always in background. An ideal garbage collector should be able to allocate the precise amount of required rememory, at the appropriate time, without wastes; it should be able to make immediately available the amount of memory occupied by an object no longer needed by the application; naturally, without interfering with the program about CPU cycles.
How does a real garbage collector work? It should make a difference between live objects, i.e. those virtually still be reached by application, and garbage objects, i.e. those can't be reached by program and therefore they are collectible. Its task is to understand if and when release that memory space busy by useless objects, making it again available for the application. So, we speak about conservatism to describe the attitude that an effective garbage collector should possess on the research of garbage objects. Therefore, it's better to keep in memory an object with a doubtful liveness rather than collect it, that is delete it from memory. Another issue on which to focus is that an automatic memory management could lead to a more-or-less accentuated fragmentation of heap. When an object is removed, it's heap is "freed", making a "hole" not always easily bridgeable by other objects. An effective garbage collector should be able to fight this fragmentation, or possibly prevent it.

What are the tasks of a garbage collector

The garbage collector begins to work if one of these conditions is satisfied:

- Ram is not enough.

- Memory used by allocated objects in heap exceeds an acceptable threshold.
- The method GC.Collect has been called (used in exceptional situations or testing).

The garbage collection operation can be decomposed on two steps: the former is garbage detection, i.e. making difference between live and garbage objects; the latter is reclamation, i.e. deleting the garbage objects, making again available their memory space for the main program.
The garbage detection can be made through two techniques: reference counting and tracing. The first is based on tracking references which point to an object; the second takes advantage of a series of memory scans to look up any live object. While the cost of the former is proportional to the amount of work carried out by the program (because it has to increment or decrement some counters for any allocation or deallocation of an object), the tracing is marked by a cost proportional to heap size, and therefore it is much more efficient.

Advantages

The correct use of garbage collector can lead to important benefits, as:

- As mentioned above, the most evident is for the developer, that should not deal with the recycling of allocated objects; the garbage collector ensures that the resources no longer used are remittances available for the application.
- It eliminates, or better reduces, the existence of dangling pointers: they are pointers that refer to memory areas that contained objects, but which have been deallocated.
- It's able to resolve several cyclic dependencies, i.e. when A points to B and B points to A.
- Some types of memory leaks, avoiding memory breakdown.

Drawbacks

Unfortunately, the same advantages of garbage collection are not always solvable by it:

- Even with a garbage collector, cyclic references are a problem, since objets need to execute some code when they are destroyed. It's impossible to determine which destructor has to be called first. It's necessary that developers clarify what should be done, even with a finalizer system.
- Memory leaks may occur even if there is a garbage collector; this could happen if the program keeps a reference to objects that have outlived their logical loop of life in the application. The presence of active references implies that the garbage collector can't detect that amount of memory as unused.
- Garbage collector consumes computing resources for tracking the memory areas used and for deciding when and how amount of memory has to been freed.
- The moment in which the garbage collection begins is not foreesable; this may produce a suddent block execution. This kind of problem are unacceptable in real-time systems or in processing of transitions.

In conclusion, it's necessary to admit that a kind of memory management is required. Looking at its set of features and drawbacks, garbage collection isn't the right approach for a generalo purpose language; even if the manual style of allocate and deallocate leads to problems, on the other side garbage collection has only one unique feature, i.e. the ability to resolve cyclic loops, that it can't actually solve correctly. Maybe in the next future a new technique will solve some of the garbage collector's drawbacks, but up to today garbage collection in generic languages has introduced more problems than it has solved.