Additional Considerations
1. Manual Memory Management (Advanced Use Cases)
While the system is designed to be automatic, there may be cases where developers need manual control over memory (e.g., for performance optimization or resource management). Adding a manual free function can provide this flexibility.

pseudo
Copy
```
object t = new Object()
t.property = "some value"
free(t)  // Manually free the object
```
2. Debugging and Profiling Tools
To help developers understand and optimize memory usage, include debugging and profiling tools in the VM:

Memory Usage Logs: Track allocations, deallocations, and reference counts.

Leak Detection: Identify objects that are not freed when they should be.

Cycle Detection: Warn about potential circular references.

pseudo
Copy
```
DEBUG_MODE = true

object t = new Object()
t.property = "some value"
free(t)

// Debug output:
// Allocated Object at 0x1234
// Freed Object at 0x1234
```
3. Thread Safety
If Amethyst supports concurrency, ensure that memory management is thread-safe. For example:

Use atomic operations for reference counting.

Protect shared data with locks or mutexes.

pseudo
Copy
```
thread {
    object t = new Object()
    t.property = "thread-safe value"
    free(t)
}
```
4. Custom Destructors
Allow developers to define custom destructors for complex objects that require special cleanup (e.g., closing files, releasing external resources).

pseudo
Copy

```
class File {
    file_handle = open_file("example.txt")

    destructor() {
        close_file(file_handle)  // Custom cleanup
        internal_free_not_consumed()
    }
}
```
5. Memory Pools

For performance-critical applications, consider adding memory pools or arena allocators to reduce fragmentation and improve allocation/deallocation speed.

pseudo
Copy
```
memory_pool pool = create_memory_pool()

object t = pool.allocate(Object)
t.property = "pool-allocated value"
pool.free(t)
```
Best Practices

Minimize Global Variables:

Encourage developers to limit the use of global variables, as they can complicate memory management and lead to leaks.

Use References Wisely:

Avoid unnecessary references to large objects, as they can delay deallocation.

Break Circular References:

If circular references are unavoidable, provide tools to detect and break them.

Test Thoroughly:

Ensure the memory management system is rigorously tested for edge cases, such as:

Deeply nested objects.

Circular references.

Concurrent access.

Document the System:

Provide clear documentation on how memory management works, including:

When destructors are called.

How reference-based reading and copy-on-write work.

How to use manual memory management (if supported).

Final Summary
Core Features:
Destructor Chain: Automatically frees objects and their properties when they go out of scope.

Reference-Based Reading: Access properties without duplicating data.

Copy-on-Write: Create new instances of data when modifying referenced values.

Scope-Based Freeing: Objects are freed when they go out of scope, but properties are preserved if still referenced.

Perks:
No Garbage Collector: Reduces runtime overhead.

Predictable Behavior: Memory is freed as soon as it's no longer needed.

Minimized Duplication: References avoid unnecessary data copying.

Additional Tools:
Manual free: For advanced memory management.

Debugging Tools: To track memory usage and detect leaks.

Thread Safety: For concurrent applications.

Custom Destructors: For complex cleanup tasks.

Memory Pools: For performance optimization.

Example Workflow (Pseudo-Code)
pseudo
Copy
```
// Global variable
global_value = 0

{
    // Create an object
    object t = new Object()
    t.property = 2

    // Reference t.property
    global_value = t.property  // global_value references t.property

    // Modify global_value (copy-on-write)
    global_value = 2 + 4  // global_value now stores its own data (6)
                          // t.property is no longer referenced by global_value
}  // t goes out of scope, destructor is called

// t.property is still referenced by global_value
print(global_value)  // Output: 6
```

```
// Free global_value when it goes out of scope
free(global_value)  // t.property is finally freed
```

Conclusion

With this system, Amethyst has a robust and efficient memory management system that balances automatic cleanup with manual control. By addressing edge cases, providing debugging tools, and following best practices, you’ve created a system that is both powerful and user-friendly. Congratulations on designing such a well-thought-out system!