V0.4 - 01/07/2023

In the context of this feature, a   is a function called before an object's lifetime ends. The   of an object refers to the period during which it is accessible and valid.

Introducing the new qualifier   , which can be used to declare a type that requires the invocation of a   before the end of its lifetime.

For example, we can declare   as follows:

To assign ownership to the result of   , we need to assign it to an variable:

The compiler will generate an error message stating that the object 'p' was not moved or destroyed.

To address this, we define a destructor for the object by declaring   as follows:

The compiler informs us that ‘p’ cannot be moved to implicitly and suggests using before the argument.

By explicitly using , we indicate that the variable has been moved or destroyed at the caller’s side.

The reason for that I want to make clear at the caller side that a variable has been moved/destroyed.

However, for certain functions with obvious move semantics based on their names, we can use the attribute to make the usage of optional:

I hope this brief tour provides you with a glimpse of what I aim to achieve.

With the introduction of the qualifier, certain changes in the type system are necessary. Similar to being cautious and explicit when moving a variable into a function, we should adopt the same approach for assignments.

Consider this sample

In this case, assigning into leads to a memory leak of the object and a double free of the object.

To address this issue, the compiler will check the end of lifetime of before the assignment.

The move assignment can be only used if both variables are owner.

The syntax is:

There is nothing especial on this assignment compared with the normal assignment. The only difference is that the intention is explicit.

After any move (assignment or function argument), the object transitions into an uninitialized state. This state is only for static analysis purposes and has no runtime implications.

Returning to our sample:

The compiler will complain that was not moved or destroyed before the assignment. To resolve this, we modify the code as follows:

If you attempt to use after moving it, the compiler will issue a warning about using an uninitialized variable.

In some cases, it may not be possible to determine if an object is initialized or not. For such scenarios, the compiler suggests using options like assertions, the attribute, or destroying the object before assignment.

For instance:

In this case, the compiler will display a message stating that the state of cannot be determined and provides suggestions for handling this situation:

This is how to fix it

or

or

We can assign a non owner to a owner.

In this situation the ownership is not transfered. The same think happens when we can function arguments that are not owners.

We can apply the qualifier to structs, unions, and enums as well:

This syntax works. However, if we forget to include the qualifier for an object that requires a destructor, we have a leak.

To address this, an additional syntax is provided for tagged objects:

By using this syntax, the object is qualified as by default:

So far, the pointer samples have used . Now, let’s consider the following situation:

Here, is an pointer to an object. In this case, the pointer is the owner of both the memory and the object. Moving the pointer will transfer both responsibilities, resulting in the destruction of the object and the memory.

Consider this sample

This code demonstrates the same behavior as the following code:

Both scenarios require the same checks, indicating that there is nothing special about the destructors.

The compiler needs to check each member of the struct individually:

Having this logic in a specialized function makes the compiler's job easier, as the flow analysis becomes simpler. It's important to assist the compiler in order to leverage its capabilities.

Another detail to note is that when we free an pointer using , the compiler assumes that we are destroying the memory and not the pointed object.

If the pointed object is also an , the compiler checks if the object is destroyed first. In the provided sample, was the only member of the struct, so it was safe to call on .

Consider:

This code is correct and works as expected.

However, if we want to pass the struct using a pointer like:

The problem arises when we pass an pointer. The compiler assumes that we want to destroy both the object and the memory. However, in this case, the object is on the stack, and we only want to destroy the object, not to free the memory.

To address this, a qualifier called is introduced, which can only be used for pointers:

This qualifier indicates that the pointer is the owner of the object but not the owner of the memory.

Returning an variable is the same as moving it. The design decision here was not require the keyword.

As expected arrays and pointer are related.

The qualifier can be placed inside the array together with the array size:

We can also pass an pointer:

By convention, passing an pointer to an array destructor will not transfer ownership of the memory, just of the pointed object.

To destroy both the array and the memory, we can use:

Let’s examine how these rules can help with and .

In this scenario, we encounter a problem because not all control paths call the destructor. The compiler would emit a warning in such cases.

However, the code is correct because we don’t need to and cannot call on a null pointer.

To address this, null checks need to be implemented in the static analyzer. The compiler will not emit warnings if it can prove that an variable is empty or uninitialized at the end of its lifetime.

When initialization needs to be checked using a result code, we don’t have semantics to provide the necessary information to the compiler. In this case, an annotation `

In this case, an annotation is needed to inform the compiler that the variable is initialized, and an annotation is needed to inform the compiler that the variable is uninitialized.

Implement this in cake. http://thradams.com/cake/index.html The hard part is the flow analysis.

This feature aims to provide ownership checks in C by introducing the qualifier. It ensures that objects are properly destroyed or moved before their lifetime ends, preventing memory leaks and use-after-free errors. The compiler assists in detecting potential issues and suggests necessary changes to the code. By leveraging ownership checks, developers can write safer and more reliable code in C.

I have placed the motivation section at the end, considering that memory safety guarantees are already a widely discussed topic and the motivation becomes apparent as readers delve into the content.

In the C programming language, manual management of resources such as memory is necessary. We rely on functions like to allocate memory and store the resulting address in a variable. To properly deallocate memory when it is no longer needed, we must use the address returned by and call the function.

Consequently, the variable holding the memory address is considered the owner of that memory. Discarding this address without calling would result in a memory leak, which is an undesirable scenario.

Resource leaks present a significant challenge because they often remain silent problems, initially having no immediate impact on a program's behavior or causing immediate issues. These leaks can easily go unnoticed during unit tests, creating a false sense of security. It is crucial to address and track these problems early on. By doing so, we can not only prevent potential complications but also save valuable time and resources in the long run.

Moreover, these checks also help prevent occurrences of double free or use-after-free issues. While both problems typically lead to immediate failures at runtime, having preventive measures in place is highly advantageous.