V0.4 - 01/07/2023

Ownership checks for C

> *This makes you feel like you're engaging in pair programming with an attentive developer working to make the program safe.*

One minute tour

In this context, *destructor* is a function called before the end of lifetime of an object. The *lifetime* of an object refers to the period during which the object is accessible and valid.

We have a new qualifier *owner* that can be used to declare a type which requires the call of some *destructor* before the end of it's lifetime.

For instance, we can declare malloc as follows:

```
void * owner malloc(int 1);
```

We need the assign a owner variable to the result of malloc.

```
int main() {
  void * owner p = malloc(1);
}
```

then compiler will say **"object 'p' was not moved/destroyed"**

Let's create a destructor, declaring free as:

```
void free(void * owner p);
```

```
int main() {
  void * owner p = malloc(1);
  free(p); //error
}
```

But the compiler will complain again.

“p cannot be moved to free implicitly, use move before the argument”.

Adding move before the argument it works.

```
   free(move p);
```

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

However, some functions have a obvious move semantic just reading the name. So for this kind of function we have an attribute `[[implicit]]`

```
void free([[implicit]] void * owner p);
```

That makes optional the usage of *move* before the argument.

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

Move assignment

With the introduction of the owner qualifier we need some changes in the type system.

In the same way we are cautious and explicit about moving a variable into a function we are with the assignment.

Consider this sample

```
int main() {
  void * p1 = malloc(1);
  void * p2 = malloc(1);
  p1 = p2;
  free(p1);
  free(p2);
}
```

Assign p2 into p1 will create a memory leak of p1 object and double free of p2 object.

To be more explicit about this I created the move assignment

```
p1 = move p2;
```

The generated code for this assignment is identical of the normal assignment.

After any move (assignment or function argument) the object transitions into an uninitialized state. Again nothing in runtime, this is just a state for the static analyzer.

Returning to our sample

```
int main() {
  void * owner p1 = malloc(1);
  void * owner p2 = malloc(1);
  p1 = move p2;
  free(p1);
  free(p2);
}
```

Compiler will complain **“p1 not moved/destroyed” before the assignment"**.

So, this is how we can fix that.

```
int main() {
  void * owner p1 = malloc(1);
  void * owner p2 = move p1;
  free(p2);
}
```

In case you try to use `p1` after moving it the compile will complain `warning : using a uninitialized variable`.

In some cases, we will not be able to check if object is initialized or not.

```
struct X { char * owner name; };
void some_function(struct X * p) {
  p->name = strdup("new text");
}
```

The compiler will have to say

`the state of 'p->name' cannot be determinated. You can use assertion or attribute [[unitialized]] or destroy the object before the assigment`

This is what we need to do

```
  free(p->name);
  p->name = strdup("new text");
```

or

```
  assert(p->name == NULL);
  p->name = strdup("new text");
```

or

```
  [[unitialized]] p->name = move strdup("new text");
```

structs/union/enum

We can apply the `owner` qualifier like this

```
int main() {
 owner struct X x = {};
}
```

It works. However, imagine you forgot to put the qualifier in a object that needs a destructor? Then we have a leak.

For this reason, we have an extra syntax for objects with tags.

```
struct owner X {
...
}
```

The consequence is that object is `owner` qualified by default.

```
int main() {
   struct X x = {};
} //**"object 'x' was not moved/destroyed"**
```

pointers

So far, the samples with pointers were using `void *`.

Now consider this situation:

```
struct owner X { char * owner name; };
int main() {
   struct X * owner p = new_x();
}
```

p is a owner pointer to a owner object. In this case the pointer is the owner of the memory and the object. Moving the pointer will move the both responsibilities, destroy the object and the memory.

Consider this sample

```

void x_delete([[implicit]] struct X * owner p)
{
   if (p){
      p->free(name);
      free(p);
   }
}
int main() {
   struct X * owner p = new_x();
   x_delete(p);
}
```

and compare with

```
int main() {
   struct X * owner p = new_x();
   if (p){
      p->free(name);
      free(p);
   }
}
```

The checks necessary inside the `x_delete` are exactly the same of the checks necessary in `main`. This shows that there is nothing especial about the *destructors*.

What compiler will have to do check each of it's owner member separately.

```
int main() {
   struct X * owner p = new_x();
   if (p){
      free(p->name);
      free(p);
   }
}
```

Having this in a specialized function makes the life of the compiler much easier because the flow analysis is simpler. You have to help the compiler to let it help you.

One more detail, when we free the owner pointer using `void*` the compiler assumes we are destroying the memory not the pointed object. So it the pointed object is owner the compiler has to check the object is destroyed first. In this sample, `free(p->name);` was the only owner member of the struct, so it was safe to call `free` on p.

destroying structs\unions

Consider:

```
struct owner X { char * owner name; };
void x_destroy([[implicit]] struct X x)
{
   free(x.name);
}
int main() {
   struct X x;
   x_destroy(x);
}
```

This code is correct and works.

But let’s say you want to pass the struct using a pointer.

```
void x_destroy([[implicit]] struct X * p) {
   free(p->name);
}
```

The problem is that if we pass a owner pointer the compiler will think we want to destroy the object and the memory. But the object is on the stack and we need to destroy only the object.

To fix this, we have a qualifier `obj_owner` that can be used only for pointers.

```
void x_destroy([[implicit]] struct X * obj_owner p) {
   free(p->name);
}
```

It means the pointer is the owner of the object but not the owner of the memory.

Returning owner type

Returning a owner variable is the same of moving it. No need to use `move`;

```
struct list make()
{
  struct list {...};
  return list;
}
```

Owner arrays

As expected arrays and pointer are related. We place owner inside together with the array size.

```
void array_destroy(int n, struct X a[owner n])
{
}

int main()
{
  struct X a[owner 100];
  array_destroy(100, a);
}
```

We can pass owner pointer.

```
void array_destroy(int n, struct X a[owner n])
{
}

int main()
{
  struct X * owner p = calloc(100, sizeof(struct X));
  array_destroy(100, p);
  free(p);
}
```

by convention passing owner pointer to array destructor will not transfer the memory ownership.

To destroy both we use:

```
void array_delete(int n, struct X * owner p)
{
}

int main()
{
  struct X * owner p = calloc(100, sizeof(struct X));
  array_delete(100, p);
}
```

Reality check I

Let’s check if these rules can help us with `fopen/fclose`.

```
FILE* owner fopen(char const* name,char const* mode);
int fclose([[implicit]] FILE* owner f);
```

```
int main() {
  FILE * owner p = fopen("text.txt", "r");
  if (p) {
    fclose(p);
  }
}
```

We have a problem, because the not all control paths are calling the destructor and the compiler will emit an warning.

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

To solve this problem we also need null-checks in your static analyzer.

The compiler will not emit warning if it can prove that a owner variable is empty or uninitialized at the end of lifetime.

Reality check II

```
int main()
{
  FILE * owner f = NULL;
  if (fopen_s( &f,"f.txt", "r") == 0)  {
    fclose([[initialized]] f);
  }
  [[uninitialized]] f;
}
```

When the initialization must be checked using a result code, we don't have this semantics and the compiler does not know that. The solution is an annotation. [[initialized]]

What's next?

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

Motivation

> *I put the motivation at the end, because these memory safety guarantees are already a hot topic, and the motivation is already present when someone reads this.*

In C, resources such as memory are managed manually. For example, we utilize the `malloc` function to allocate memory and store the resulting address in a variable. When the memory is no longer needed, we need the address returned by `malloc` to be able to call `free`.

Therefore, the variable holding the address is considered the owner of the memory, as this address cannot be simply discarded, otherwise we have a memory leak.

Resource leaks pose a significant challenge as they tend to be silent problems that don't immediately impact a program's behavior or cause immediate issues. Moreover, they can easily go unnoticed during unit tests, creating a false sense of security. Therefore, it is absolutely crucial to address and track these problems early on. By doing so, not only can potential complications be prevented, but it can also save valuable time and resources in the long run.

This checks also prevent double free, or use after free. Both problems generally fail fast in runtime, but it is also good to have.