





Implement CppInterOp API

Exposing Memory Ownership

Kerem Şahin
6 November 2026

Mentors:
Vipul Cariappa, Aaron Jomy, Vassil Vassilev

CppInterop, cppjit and Dynamic Language Binding

Language binding tools let users call and use code from other languages. The concept is best known for letting Python code use machine-learning and image-processing libraries written in C/C++.

Binding can be done either statically or dynamically; while static binding requires glue code to carry the necessary information to bind between languages, dynamic binding extends the ability of the calling code to define and call code of the target language at runtime. However, dynamic binding requires using an interpreter of the target language to define/call code and extract information from the AST by querying.

Static vs Dynamic Binding

```
Klass* createKlass(int id, int size = -1);
Klass* createKlass(const char* name);

// bindings.cpp
m.def("createKlass", py::overload_cast<int, int>(&createKlass),
      py::arg("id"), py::arg("size") = -1,
      py::return_value_policy::take_ownership);
m.def("createKlass", py::overload_cast<const char*>(&createKlass),
      py::arg("name"), py::return_value_policy::take_ownership);
```

```
// What cppjit asks CppInterOp at runtime (simplified)

// Gives "createKlass" and its overloads
auto cands = Cpp::GetFunctionsUsingName(Cpp::GetGlobalScope(), "createKlass");
// py::overload_cast<int, int>
auto f     = Cpp::BestOverloadFunctionMatch(cands, argTypes);
// py::arg("size")
auto name  = Cpp::GetFunctionArgName(f, 1);
// = -1
auto def   = Cpp::GetFunctionArgDefault(f, 1);
// &createKlass
auto call  = Cpp::MakeFunctionCallable(f);
// Here comes my project
auto ownership = ???
```

The Problem (1/2)

cppjit should invoke the corresponding deallocator once garbage collection determines that an object's lifetime has ended. However, if the user does not set ownership, cppjit cannot reliably determine which returned objects it actually owns, which leads to a memory leak.

```
int* IntAllocator() {
  return (int*)malloc(sizeof(int));
}
```

```
def foo():
  x = gbl.IntAllocator() # cppjit does not know that it owns x
  return None # Leak: x is not deallocated
```

The Problem (2/2)

Currently, cppjit cannot distinguish between functions that return freshly heap-allocated memory and functions that return memory that should not be owned by the caller.

```
Klass* createKlass() {
    return new Klass();
}
```

```
Klass* getDefaultInstance() {
    static Klass instance;
    return &instance;
}
```

Project Description

- The main goal of the project is to extract information about the memory ownership of C++ functions. cppjit uses this information to automate memory deallocation to eliminate memory leaks.
- Example: cppjit should recognize that the object returned by this function requires manual deallocation, and identify the deallocator that is most appropriate for the object.

```
Klass* AllocKlass() {
    return new Klass;
}
```

```
-FunctionDecl <line:4:1, line:6:1> line:4:8 All
  -CompoundStmt <col:21, line:6:1>
    -ReturnStmt <line:5:5, col:22>
      -CXXNewExpr <col:12, col:22>
        -CXXConstructExpr <col:16, col:22>
```

CppInterOp Side

There are two ways to determine the memory ownership behaviour of C++ functions:

1. Indication by developer: Attributes or APINotes.
2. Analyzing the AST: Traversing the AST and looking for meaningful constructs such as return statements and variable declarations.

```
__attribute__((malloc))
Klass* AllocWithAttr();
```

```
-FunctionDecl <line:3:1, col:46> col:32
  -RestrictAttr <col:16> malloc
    -<<<NULL>>>
```

```
Klass* AllocKlass() {
    return new Klass;
}
```

```
-FunctionDecl <line:4:1, line:6:1>
  -CompoundStmt <col:21, line:6:1>
    -ReturnStmt <line:5:5, col:22>
      -CXXNewExpr <col:12, col:22>
        -CXXConstructExpr <col:16, col:22>
```

Using and Representing Information

Information gathered from attributes and analysis must be carried forward, but traditional Clang/GCC attributes are not sufficient for this purpose, since the specific method or function used for allocation must also be known in order to deallocate memory properly.

Therefore, this information is carried using an enum:

```
enum class AllocType : unsigned char {
  None,
  New,
  NewArr,
  Malloc,
  OperatorNew,
  OperatorNewArr,
  Null,  // Needed for inner logic, no usage in API
  Unknown
};
```

cppjit Side

- When a C++ function is called in cppjit, it first checks whether the function carries a memory-ownership attribute. If no memory-related attribute is present and the analyzer is enabled (disabled by default), the function’s AST is traversed and the result is cached.
- This result is used to set flags on the Python proxy of the C++ object returned by the function. This applies to both record-type and primitive-type objects. When the garbage collector determines that the object’s lifetime has ended, this result is used to call the appropriate deallocator.

How to Use - Analyzer

The memory analyzer is disabled by default; the user must set a flag to enable it.

```
import cppjit

obj = cppjit.gbl.IntAllocator() # Leak: No attribute info, nor analyzer enabled

cppjit.use_memory_analyzer(True) # Enable analyzer
obj2 = cppjit.gbl.IntAllocator() # Leak again, result was cached when analyzer was off

obj3 = cppjit.gbl.IntAllocator2() # No leak
cppjit.use_memory_analyzer(False) # Disable analyzer

obj4 = cppjit.gbl.IntAllocator2() # No leak: result from former call is cached
```

- The analyzer requires the function's definition to analyzer.
- The value returned by the analyzer is cached and successive calls to the same function account for it.

How to Use - Attribute

- Traditional compiler attributes are not sufficient, so a custom identifier pattern was created; there are two ways to carry this attribute pattern through a function's AST.
- Pattern: `cppAlloc`Allocation Type
- For example, when attribute `cppAllocOperatorNewArr` is inserted into the AST, our API recognizes it and knows that no constructor was run on the memory, therefore no destructor should be called when the object's lifetime ends.

How to Use - Attribute

- **1. Insertion by redeclaration:**
  A function can be redeclared with the same signature, and Clang will merge the declarations.
- Example:

```
#include "CppInterOp/CppInterOp.h"

namespace Cpp {
[[clang::annotate("cppAllocOperatorNew")]]
ObjectRef Allocate(DeclRef DRef,
                   size_t count);
} // namespace Cpp
```

```
#include <cstddef> // Needed for size_t
namespace Cpp {
struct ObjectRef;
struct DeclRef;
[[clang::annotate("cppAllocOperatorNew")]]
ObjectRef Allocate(DeclRef DRef,
                   size_t count);
} // namespace Cpp
#include "CppInterOp/CppInterOp.h"
```

```
[[clang::annotate("cppAllocNew")]]
int* myIntAllocator();
#include "myAllocs.h"
```

How to Use - Attribute

1. Insertion by redeclaration:

Attributes can also be inserted into multiple functions at once.

- Example:

```
#include "CppInterOp/CppInterOpTypes.h"
#pragma clang attribute push([[clang::annotate("cppAllocNew")]], apply_to = function)
namespace Cpp {
ObjectRef Allocate(DeclRef DRef, size_t count);
ObjectRef Construct(DeclRef DRef, void* arena, size_t count);
}
#pragma clang attribute pop
#include "CppInterOp/CppInterOp.h"
```

How to Use - Attribute

2. Insertion by APINotes:

Attributes can also be inserted using Clang's APINotes by writing to the SwiftReturnOwnership field. There is no need to copy the exact function signature — only the function names and the header, referenced by its relative or absolute path, are required.

The interpreter must be given the `-fmodules -fimplicit-module-maps -fapinotes-modules` flags to use APINotes using environment variables. These flags cannot be set after the interpreter is created, although include paths can be added afterward.

How to Use - Attribute

2. Insertion by APINotes (Example):

MyAPINotes/module.modulemap

```
module MyHeader {header "/home/.../root-project/.../SomeHeader.h"}
module AnotherHeader {header "../include/SomeAnotherHeader.h"}
```

MyAPINotes/MyHeader.apinotes

```
Name: MyHeader
Functions:
  - Name: SomeRootFunc
    SwiftReturnOwnership: cppAllocMalloc
Tags:
    - Name: SomeRootClass
      Methods:
        - Name: SomeClassMethod
          SwiftReturnOwnership: cppAllocNew
```

MyAPINotes/AnotherHeader.apinotes

```
Name: AnotherHeader
Functions:
  - Name: SomeAnotherFunc
    SwiftReturnOwnership: cppAllocNewArr
Tags:
    - Name: SomeAnotherStruct
      Methods:
        - Name: SomeAnotherMethod
          SwiftReturnOwnership: cppAllocNew
```

```
import cppjit
cppjit.add_include_path("/home/.../MyAPINotes/")
cppjit.include("../include/SomeAnotherHeader.h")
```

How to Use - Attribute

- There are also attributes that are used to label a function's deallocation behaviour for each parameter using CppInterOp's `DeallocType` values.
- Indexing starts from one, following Clang's convention, and implicit **this** parameter can not be labeled even though it occupies the 1st index at the argument list.

```
- Name: testDeallocFree
  SwiftReturnOwnership: cppDeallocFree_1
- Name: testDeallocDelete
  SwiftReturnOwnership: cppDeallocDelete_1
- Name: testDeallocDeleteArr
  SwiftReturnOwnership: cppDeallocDeleteArr_1
- Name: testDeallocNone
  SwiftReturnOwnership: cppDeallocNone_1
- Name: testDeallocMulti
  SwiftReturnOwnership: "cppDeallocDelete_1 cppDeallocFree_2"
```

Result: Before & After

```
template <typename T, typename ...Args>
[[clang::annotate("cppAllocNew")]] T* Create(Args&&... args) {
    return new T(args...);
}
```

Before:

```
def foo():
  x = gbl.Create[int]()
  assert x.__python_owns__ == False
  return None # leak
```

After:

```
def foo():
  x = gbl.Create[int]()
  assert x.__python_owns__ == True
  return None # no leak
```

Manually set ownership of every object:

Before:

```
def foo():
  x = gbl.Create[int]()
  # manually set ownership
  x.__python_owns__ = True
  return None # no leak
```

After:

```
def foo():
  # Ownership info set in C++,
  # for all objects returned by this function
  x = gbl.Create[int]()
  # automatically set ownership
  assert x.__python_owns__ == True
  return None # no leak
```

Possible Problem: Double Free

If a chunk of memory is deallocated explicitly by the user, and cppjit also decides to deallocate it because of attributes or the analyzer, this results in a double free.

```
template <typename T, typename ...Args>
[[clang::annotate("cppAllocNew")]] T* Create(Args&&... args) {
  return new T(args...);
}

template <typename T>
void Release(T* obj) { delete obj; }
```

```
def foo():
  x = gbl.Create[int]() # python owns, so when the function ends it is collected
  gbl.Release[int](x) # x is already freed here
  return None # cppjit also tried to delete x again, this is double free
```

How to Solve It?

Every function call is checked to update the state of ownership for each object using analyzer and attributes. For example if a function with `ownership_holds` attribute is called, the ownership of the object that given as an argument to the corresponding parameter will be given back.

```
void __attribute__((annotate("cppDeallocDelete", 1)))
ReleaseKlass(Klass* p);
```

```
import cppjit
cppjit.include("KlassHeader.h")
obj = cppjit.gbl.CreateKlass()

# Since ReleaseKlass takes the ownership of obj, there will not be any deallocation
# inserted by the time scope ends/GC collects the obj.
cppjit.gbl.ReleaseKlass(obj)
```

How Objects are Handled?

Each object carries its ownership state, and allocation details like whether the constructor was called or whether it points to an array.

This information is gathered by inspecting each function callee and their ownership behaviour. When GC collects an object, the corresponding deallocation function is inserted if the object is still owned by the Python side.

```
kIsOwner = 0x00002,       // Python instance owns C++ object/memory
kIsArrayAlloc = 0x04000, // represents a heap allocated array of objects
kIsNoConstruct = 0x08000, // represents constructor is not called in the allocation
kIsMalloc = 0x10000, // is allocated with malloc
```

Thanks for Listening