How to use C++ classes with ctypes?

Asked 11 years, 10 months ago Active 4 years, 1 month ago Viewed 55k times

61

27

I'm just getting started with ctypes and would like to use a C++ class that I have exported in a dll file from within python using ctypes. So lets say my C++ code looks something like this:

```
class MyClass {
  public:
    int test();
...
```

I would know create a .dll file that contains this class and then load the .dll file in python using ctypes. Now how would I create an Object of type MyClass and call its test function? Is that even possible with ctypes? Alternatively I would consider using SWIG or Boost.Python but ctypes seems like the easiest option for small projects.

c++ python ctypes

Share Edit Follow Flag

edited Jan 23 '15 at 18:57
Deduplicator
42.4k 6 61 105

asked Oct 23 '09 at 20:51
jörg
3,061 3 18 12

4 Answers

Active Oldest Votes

52

Besides Boost.Python(which is probably a more friendly solution for larger projects that require one-to-one mapping of C++ classes to python classes), you could provide on the C++ side a C interface. It's one solution of many so it has its own trade offs, but I will present it for the benefit of those who aren't familiar with the technique. For full disclosure, with this approach one wouldn't be interfacing C++ to python, but C++ to C to Python. Below I included an example that meets your requirements to show you the general idea of the extern "c" facility of C++ compilers.

```
//YourFile.cpp (compiled into a .dll or .so file)
#include <new> //For std::nothrow
//Either include a header defining your class, or define it here.

extern "C"  //Tells the compile to use C-linkage for the next scope.
{
    //Note: The interface this linkage region needs to use C only.
    void * CreateInstanceOfClass( void )
    {
        // Note: Inside the function body, I can use C++.
        return new(std::nothrow) MyClass;
    }

    //Thanks Chris.
    void DeleteInstanceOfClass (void *ptr)
    {
         delete(std::nothrow) ptr;
    }

    int CallMemberTest(void *ptr)
    {

        // Note: A downside here is the lack of type safety.
        // You could always internally(in the C++ library) save a reference to
all
        // pointers created of type MyClass and verify it is an element in that
        //structure.
        //
        // Per comments with Andre, we should avoid throwing exceptions.
        try
        {
            MyClass * ref = reinterpret_cast<MyClass *>(ptr);
            return ref->Test();
        }
        catch(...)
        {
```

You can compile this code with

```
gcc -shared -o test.so test.cpp
#creates test.so in your current working directory.
```

In your python code you could do something like this (interactive prompt from 2.7 shown):

```
>>> from ctypes import cdll
>>> stdc=cdll.LoadLibrary("libc.so.6") # or similar to load c library
>>> stdcpp=cdll.LoadLibrary("libstdc++.so.6") # or similar to load c++ library
>>> myLib=cdll.LoadLibrary("/path/to/test.so")
>>> spam = myLib.CreateInstanceOfClass()
>>> spam
[outputs the pointer address of the element]
>>> value=CallMemberTest(spam)
[does whatever Test does to the spam reference of the object]
```

I'm sure Boost.Python does something similar under the hood, but perhaps understanding the lower levels concepts is helpful. I would be more excited about this method if you were attempting to access functionality of a C++ library and a one-to-one mapping was not required.

For more information on C/C++ interaction check out this page from Sun: http://dsc.sun.com/solaris/articles/mixing.html#cpp_from_c

Share Edit Follow Flag

edited Aug 15 '11 at 2:18

answered Aug 15 '11 at 1:44

AudaAero

531 4 4

5 Indeed, this is a little tedious, but it works. You'll specifically want to watch out for exceptions though. I don't think it's safe to assume the `ctypes` module handles C functions that throw exceptions very well. In particular, the `return new MyClass;` statement is very dangerous

43

The short story is that there is no standard binary interface for C++ in the way that there is for C. Different compilers output different binaries for the same C++ dynamic libraries, due to name mangling and different ways to handle the stack between library function calls.

So, unfortunately, there really isn't a portable way to access C++ libraries *in general*. But, for one compiler at a time, it's no problem.

This blog post also has a short overview of why this currently won't work. Maybe after C++0x comes out, we'll have a standard ABI for C++? Until then, you're probably not going to have any way to access C++ classes through Python's `ctypes`.

Share Edit Follow Flag

answered Oct 23 '09 at 21:59

Mark Rushakoff

229k 43 391 392

15

The answer by AudaAero is very good but not complete (at least for me).

On my system (Debian Stretch x64 with GCC and G++ 6.3.0, Python 3.5.3) I have segfaults as soon has I call a member function that *access a member value* of the class. I diagnosticated by printing pointer values to stdout that the void* pointer coded on 64 bits in wrappers is being represented on 32 bits in Python. Thus big problems occurs when it is passed back to a member function wrapper.

The solution I found is to change:

```
spam = myLib.CreateInstanceOfClass()
```

Into

```
Class_ctor_wrapper = myLib.CreateInstanceOfClass
Class_ctor_wrapper.restype = c_void_p
spam = c_void_p(Class_ctor_wrapper())
```

So two things were missing: setting the return type to c_void_p (the default is int) **and** then creating a c_void_p object (not just an integer).

I wish I could have written a comment but I still lack 27 rep points.

Share Edit Follow Flag

edited Jun 20 '20 at 9:12
Community ♦
1 1

answered Mar 5 '17 at 14:46
Gabriel Devillers
1,812 2 21 37

3

Extending AudaAero's and Gabriel Devillers answer I would complete the class object instance creation by: `stdc=c_void_p(cdll.LoadLibrary("libc.so.6"))` using `ctypes c_void_p` data type ensures the proper representation of the class object pointer within python.

Also make sure that the dll's memory management be handled by the dll (allocated memory in the dll should be deallocated also in the dll, and not in python)!

Share Edit Follow Flag

answered Aug 1 '17 at 7:54

Elektrone Motyo

41 2