MQF 633 C++ for Financial Engineering

Lecture 9: C++ with Python and Database

Part I: Working with Python Using Ctypes

Python **ctypes** is a foreign function interface (FFI) library for Python that allows calling functions and using data types from shared libraries written in other languages, such as C. It provides mechanisms for calling functions dynamically, loading DLLs or shared libraries, and passing data back and forth between Python and C (C++) code.

By using **ctypes**, Python programs can access functions and data structures defined in C/C++ libraries without having to write custom C/C++ extension modules. This can be useful when you want to interact with existing C/C++ libraries from your Python code or when you need to work with low-level system functions that are not directly accessible from Python.

More details of **ctypes** documentation can be found in below link:
https://docs.python.org/3/library/ctypes.html

Here's a simple example demonstrating how **ctypes** can be used to call a function from a shared library.
It contains a few steps in general:

1. Create C/C++ functions, compile it into .so, shared library object. In this lecture we use .so instead of dll. The difference between the two lies in below link.
   https://www.jwhitham.org/2017/10/dll.html
2. Create a python file, and import **ctypes**, and then load C/C++ function from shared library
3. Make function call from python into C/C++ function in library.

First Example of Hello World

Let's have a look on famous example of "hello_world" program.
Create below in hello_world.cpp

```
#include <stdio.h>

extern "C" void hello_world()
{
printf("Hello World ...\n");
}
```

Compile it using command of below to create a shard object library file with .so.

```
g++ -shared -o testlib.so -fPIC hello_world.cpp
```

This will create a file with name of testlib.so in your working path. Then create below hello_world.py with code of:

```
import ctypes
import os

clib = ctypes.CDLL(os.getcwd() + "/testlib.so")
hello = clib.hello_world
hello()
```

The result:

```
david@Yans-MacBook-Air python_ctypes % /usr/local/bin/python3 /Users/david/Documents/GitHub/Cplusplus/src/code_L9/python_ctypes/
hello_world.py
Hello World ...
```

Take Note

- In this example, we are putting `extern "C"` before c++ function return type. This is due to:

extern "C" makes a function-name in C++ have C linkage (compiler does not mangle the name) so that client C code can link to (use) your function using a C compatible header file that contains just the declaration of your function. Your function definition is contained in a binary format (that was compiled by your C++ compiler) that the client C linker will then link to using the C name.
Since C++ has overloading of function names and C does not, the C++ compiler cannot just use the function name as a unique id to link to, so it mangles the name by adding information about the arguments. A C compiler does not need to mangle the name since you can not overload function names in C. When you state that a function has extern "C" linkage in C++, the C++ compiler does not add argument/parameter type information to the name used for linkage.

- We are using ctype.CDLL() function to load C/C++ function into python ctypes namespace. This in fact is a function ptr. To make function call, you have to use () with function name, re-defined in python.

If using Windows, then it is possible to directly load windows library. In below example, msvcrt is a pre-compiled windows library which contains printf() function. We then can directly make C function call in python after loading this library.

```
from ctypes import *

# this example can be run in windows only
libc = cdll.msvcrt
printf = libc.printf
printf(b"Hello, %s\n", b"World!")
printf(b"%d bottles of beer\n", 42)
```

Data Types Supported in Python Ctypes

We already know that C/C++ is type strict and python is not. In order to pass in/out argument and result between Python and C/C++, the type must be compatible and declared explicitly. For all most all types in C/C++, there is corresponding types in python **ctypes** to be mapped. Refer to below table.

Take note, there is no std::string here, since string in C++ is a STL template class. We need to use char * when working with Python with C.

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_time_t	time_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (NUL terminated)	bytes object or None
c_wchar_p	wchar_t* (NUL terminated)	string or None
c_void_p	void*	int or None

Refer to datatype.py

Basic types

```
import ctypes as ct
# value_1 variable stores an integer of type Ctypes (not a regular integer)
int_value = ct.c_int(10)
# printing the type and what int_value holds
print(int_value, int_value.value)

# storing a ctypes long value
long_value = ct.c_long(10)
print(long_value, long_value.value)

# creating a ctypes float variable
float_value = ct.c_float(15.25)
print(float_value, float_value.value)

# creating a ctypes double variable
double_value = ct.c_double(85.69845)
print(double_value, double_value.value)

# creating a ctypes char variable
str_value = ct.c_char(b'c')
print(str_value, str_value.value)
```

Take note:

The b prefix explicitly tells Python that you are creating a byte literal. Bytes objects are used when you need to work with the actual sequence of bytes that make up data. Interfacing with C libraries (like in your **ctypes** example, where C expects raw memory/bytes).

```
# creating a ctypes wchar variable
str_value1 = ct.c_wchar('c')
print(str_value1, str_value1.value)
```

```
# creating a ctypes char * variable
str_value2 = ct.c_char_p(b"cat")
print(str_value2, str_value2.value)
```

```
# creating a ctypes wchar * variable
str_value3 = ct.c_wchar_p("test")
print(str_value3, str_value3.value)
```

Pointer Type

```
# using pointer() method we are pointing to the value_1 variable and storing it in ptr
value_1 = ct.c_int(20)
ptr = ct.pointer(value_1)
print("Contents of value_1 : ", value_1)
print("Real value stored in value_1 : ", value_1.value)
print("Address of value_1 : ", id(value_1.value))

# If we want to print the contents of a pointer type variable then need to use .contents
```

```
# otherwise only writing the variable is enough like above
print("Contents of ptr variable : ", ptr.contents)

# To print the value stored in the address pointed by that pointer variable
# we need to use .value after .contents
print("The value at which ptr points at : ", ptr.contents.value)

# Printing the address of the value to which the ptr variable points at
print(
"Address of that value which is pointed and stored in ptr : ",
id(ptr.contents.value),
)
```

Take note

- from these example we can see that all **ctype** variable we define in python is object type, need to use **.value** to get the value store in object.
- we can use pointer() function on object to assign value address to pointer

Python Object Is Immutable

```
from ctypes import *

# we change the value of p_s object, and we noticed the address changed also, this is due to python object is immutable
s = "Hello, World"
print("before change: ", id(s), s)
s = "Hello, World Changed!"
print("after change: ", id(s), s)
```

```
p_s = c_wchar_p(s)
print("before change: ", p_s, p_s.value)
p_s.value = "Hi, there"
print("after change: ", p_s, p_s.value)
```

To achieve “mutable” behaviour in python **ctypes**, we need to allocate memory buffer, this is almost like allocate memory in heap in C++. Refer to immutable.py for example.
Take note:

- The **id()** built-in function in Python returns the "identity" of an object. This identity is guaranteed to be unique and constant for the object during its lifetime. Essentially, id(variable) gives you the memory address where the object that the variable currently refers to is stored. Use **hex()** to change long int into hexadecimal format.
- The **repr()** function in Python is a built-in function that returns a "developer-friendly" or "official" string representation of an object. Its primary goal is to produce a string that, when passed to the eval() function, would ideally recreate the original object.

```
# to achieve mutable behaviour we need to use below function.
```

```
# this is quite essential when we like to receive C/C++ function return result

# create a 3 byte buffer, initialised to NUL bytes
p = create_string_buffer(3)
print(sizeof(p), repr(p.raw), repr(p.value))

# create a buffer containing a NUL terminated string
p = create_string_buffer(b"Hello")
print(sizeof(p), repr(p.raw), repr(p.value))

p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
print(id(p), repr(p.raw), repr(p.value))
p.value = b"Hi"
print(id(p), repr(p.raw),repr(p.value))
```

Passing Variable Between Python and C/C++

- Remember that data types in C and python is **different**, so some of basic types can be parsed directly through **ctypes**, most of them are not
- Types directly parsing from python to C: **None, int, bytes** and **Unicode string**

None, integers, bytes objects and **(unicode) strings are the only native Python objects that can directly be used as parameters in these function calls.** None is passed as a C NULL pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (char* or wchar_t*). Python integers are passed as the platforms default C int type, their value is masked to fit into the C type.

Function prototypes

Arg type: It is possible to specify the required argument types of functions exported from DLLs by setting the argtypes attribute. argtypes must be a sequence of C data types

```
printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10,  2.2)
```

Return type: by default functions are assumed to return the C int type. Other return types can be specified by setting the restype attribute of the function object.

Here is a more advanced example, it uses the strchr() function, which expects a string pointer and a char, and returns a pointer to a string. The strchr() function returns a pointer to the first occurrence of c that is converted to a character in string. The function returns NULL if the specified character is not found.

```
strchr = libc.strchr
strchr.restype = c_char_p
strchr.argtypes = [c_char_p, c_char]
print(strchr(b"abcdef", b"d")) # return b’def’
```

```
print(strchr(b"abcdef", b"x")) # return None
strchr(b"abcdef", b"d")
```

Passing Simple Types

Refer to simple_func.py and simple_func.cpp.

```
#include <stdio.h>

extern "C" int c_sum_int(int arg1, int arg2)
{
// Return the sum
int result = arg1 + arg2;
printf("sume is %d:\n ", result);
return result;
}
extern "C" double c_sum_double(double arg1, double arg2)
{
// Return the sum
double result = arg1 + arg2;
printf("sume is %f:\n ", result);
return result;
}
```

```
extern "C" void display(char *str)
{
printf("%s\n", str);
}
extern "C" void char_increment(char *str)
{
for (int i = 0; str[i] != 0; i++)
{
str[i] = str[i] + 1;
}
}
extern "C" int sum_arr(int *arr, int n)
{
int sum = 0;
for (int i = 0; i < n; i++)
{
sum += arr[i];
}
return sum;
}
```

Compile this using

```
g++ -shared -o simple_func.so -fPIC simple_func.cpp
```

Passing Pointer Types

In below example, using C to show the parsing pointer type from python. Refer to Pointer.c and Pointer.py for example.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *alloc_memory(void)
{
char * str = malloc(100 * sizeof(char));
str = strdup("Hello World");
printf("Memory allocated...\n");
return str;
}
void free_memory(char *ptr)
{
printf("Freeing memory...\n");
free(ptr);
}
char *str_concatenate(char *str1, char *str2)
{
char *str;
str = malloc(100 * sizeof(char));
```

```
printf("Memory allocated...%s\n", str);
memccpy(str, strcat(str1, str2), 0, 100);
return str;
}
```

Compile this using
`gcc -shared -o pointer.so -fPIC pointer.c`

Refer to Python file pointer.py

```
import os
import ctypes as ct

clib = ct.CDLL(os.getcwd() + "/pointer.so")

# example 1
if True:
c_strfree = clib.free_memory
c_strfree.argtypes = [ct.c_char_p]
c_strfree.restype = ct.c_void_p
#str = ct.c_char_p(b"this ")
#c_strfree(str)

alloc_func = clib.alloc_memory
```

```
alloc_func.restype = ct.POINTER(ct.c_char)
str_ptr = alloc_func()
str = ct.c_char_p.from_buffer(str_ptr)
print(str.value)
c_strfree(str_ptr)
```

```
# exampe 2
if True:
c_strcon = clib.str_concatenate
c_strcon.argtypes = [ct.c_char_p, ct.c_char_p]
c_strcon.restype = ct.POINTER(ct.c_char)
```

```
str1 = ct.c_char_p(b"this ")
str2 = ct.c_char_p(b"a dog")
result_ptr = c_strcon(str1, str2)
print(result_ptr, hex(id(result_ptr)), result_ptr.contents.value)
str = ct.c_char_p.from_buffer(result_ptr)
print(str.value)
c_strfree(result_ptr)
```

Take Note

- For dynamic allocated memory in C++, the return type should be pointer type.
- If memory is allocated in C/C++, then it has to be free in C/C++, otherwise leads to memory leak
- To get the value for variable which memory allocated by C/C++, need to use function ctypes.type.form_buffer(poitner*)

Parsing Array

Let have a look on an example of how to pass array from python to C.
In file simple_func.cpp

```
extern "C" int sum_arr(int *arr, int n)
{
int sum = 0;
for (int i = 0; i < n; i++)
{
sum += arr[i];
}
return sum;
}
```

In Python we have

```
c_sum_arr = clib.sum_arr
c_sum_arr.argtypes = [ct.POINTER(ct.c_int32), ct.c_int]
c_sum_arr.restype = ct.c_int

# exampe sum of int array
array = (ct.c_int * 10)()
for i in range(10):
array[i] = i
```

```
sum = c_sum_arr(array, len(array))
print(sum)
```

Parsing or Calling Struct

In real-life example, it is very useful to export structure coded in C into python, and work on the data of the structure, or even change it. But structure, like class is a user-defined type in C. let's see an example as below:

```
struct Point
{
int x;
int y;
};

struct PointArray
{
struct Point points[3];
};
```

```
void printPoint(struct Point p)
{
printf("%d %d\n", p.x, p.y);
}
```

```
struct Point getPoint1()
```

```
{
struct Point temp;
temp.x = 50;
temp.y = 100;
return temp;
}
```

```
struct Point *getPoint2()
{
struct Point *temp;
temp->x = 50;
temp->y = 10;
return temp;
}
```

```
void printPointArray(struct PointArray pa)
{
for (int i = 0; i < 3; i++)
{
printf("%d %d\n", pa.points[i].x, pa.points[i].y);
}
}
```

In Python

```
import os
import ctypes as ct

clib = ct.CDLL(os.getcwd() + "/structure.so")
```

```
class Point(ct.Structure):
_fields_ = [("x", ct.c_int),
("y", ct.c_int)]
# Create a Point Object and pass to C/C++
p1 = Point(10, 20)
clib.printPoint(p1)
```

```
# get structure from C/C++
clib.getPoint2.restype = ct.c_void_p
p2 = Point.from_address(clib.getPoint2())
print(p2.x, p2.y)
```

```
# free memory
clib.free_point(ct.byref(p2))
```

Parsing Class

Interfacing C++ classes with Python using **ctypes** is a common challenge because **ctypes** is primarily designed for interfacing with C-style APIs. C++ introduces features like name mangling, method overloading, constructors/destructors, and virtual functions, which **ctypes** doesn't inherently understand.
To "parse" or interact with a C++ class from Python using **ctypes**, you typically need to create a C-style wrapper layer around your C++ class. This wrapper exposes C functions that handle object creation, method calls, and destruction, which **ctypes** can then easily interface with.

In myclass.h we have below class.

```
// myclass.h

#pragma once


class MyClass {

public:

    MyClass(int val);

    int get_value() const;

    void set_value(int val);

private:

    int value;

};
```

In wrapper.cpp we have below code.

```
// wrapper.cpp
#include "myclass.h"

extern "C" {
    MyClass* MyClass_new(int val) {
        return new MyClass(val);
    }
    int MyClass_get_value(MyClass* obj) {
        return obj->get_value();
    }
    void MyClass_set_value(MyClass* obj, int val) {
        obj->set_value(val);
    }
    void MyClass_delete(MyClass* obj) {
        delete obj;
    }
}
```

The extern "C" block is crucial.
It tells the C++ compiler to use C-style linkage for the functions declared within it. This prevents C++ name mangling, ensuring that the function names in the compiled shared library are simple and predictable (e.g., create_calculator, calculator_add), which **ctypes** can easily find.

In myclass.cpp it contains the implementation of my class.

```
// myclass.cpp

#include "myclass.h"


MyClass::MyClass(int val) : value(val) {}


int MyClass::get_value() const

{

   return value;

}


void MyClass::set_value(int val)

{

   value = val;

}
```

Compiling these files.
Example are using Mac: generate the object file and then link into one .so file.

```
g++ -c myclass.cpp wrapper.cpp
g++ -shared -o libmyclass.so myclass.o wrapper.o
```

In python, we create myclass.py.

```
import ctypes

# Load the shared library
lib = ctypes.CDLL('./libmyclass.so')

# Set return and argument types
lib.MyClass_new.argtypes = [ctypes.c_int]
lib.MyClass_new.restype = ctypes.c_void_p
lib.MyClass_get_value.argtypes = [ctypes.c_void_p]
lib.MyClass_get_value.restype = ctypes.c_int
lib.MyClass_set_value.argtypes = [ctypes.c_void_p, ctypes.c_int]
lib.MyClass_delete.argtypes = [ctypes.c_void_p]

# Python wrapper class
class MyClass:
    def __init__(self, val):
```

```
        self.obj = lib.MyClass_new(val)
    def get_value(self):
        return lib.MyClass_get_value(self.obj)
    def set_value(self, val):
        lib.MyClass_set_value(self.obj, val)
    def __del__(self):
        lib.MyClass_delete(self.obj)


# 🧪 Usage
mc = MyClass(42)
print(mc.get_value())  # 42
mc.set_value(100)
print(mc.get_value())  # 100
```

Summary

How it works (the "parsing" aspect):

1. C-Style Wrapper: The core idea is that ctypes cannot directly understand C++ classes and their member functions (due to name mangling and object-oriented concepts not being directly exposed in a flat C API). So, you create global C functions (using extern "C") that take a void* (which will represent the this pointer of your C++ object) as their first argument.
2. Object Creation (my class):
    - This C function uses new to create an actual my class C++ object.
    - In Python, ctypes.c_void_p is used to represent this generic pointer.

3. Method Calls (calculator_add, etc.):
    - These C functions take the void* (the object's address) and any other necessary arguments.
    - You then call the actual C++ member function on this pointer (e.g., myObject->setValue(a)).
    - The return value is a simple C type that ctypes can handle (e.g., int).
4. Object Destruction (destroy_myclass):
    - This C function takes the void* pointer and uses delete to properly deallocate the C++ object, preventing memory leaks.
    - The MyClass.__del__ method in Python is crucial here to ensure that when your Python MyClass object is garbage collected, the underlying C++ object is also correctly destroyed.
5. Python Wrapper Class (PyCalculator):
    - This Python class wraps the ctypes calls, making the interaction with the C++ object feel more "Pythonic."
    - It manages the c_void_p (the pointer to the C++ object) internally.
    - It handles the creation and destruction (via __del__) of the C++ object.

This approach effectively provides a C-compatible interface to your C++ class, allowing ctypes to bridge the gap. While ctypes is powerful for C integration, for complex C++ libraries, tools like **PyBind11** or **SWIG** are often preferred as they automate much of this wrapper code generation. However, for simpler class interactions, ctypes is a perfectly viable and lightweight solution.

Part II: Working with Database

An overview of Database

1. Microsoft SQL Server

   Microsoft SQL Server best features
   - Achieve high performance with Microsoft SQL Server, which handles complex queries efficiently
   - Ensure data safety with advanced security features
   - Scale your database with SQL Server easily, handling large databases and complex transactions as your business grows

   Microsoft SQL Server limitations
   - Quite complex for beginners
   - Full-featured versions expensive for small businesses

2. Oracle Database
   - Similar to MS SQL Server, this is SQL query based logic Table RDMS developed by Oracle. It is more efficient in query but more complex for beginner.

3. PostGreSQL

   Pro:
   - Leverage its open-source nature for cost-effective data management

- Utilise advanced data types and performance optimisation features
- Customise it to align with specific business requirements
- Secure sensitive business data effectively with robust security features

Con:

- Overwhelming for beginners or small teams without dedicated database administrators
- Doesn't match the performance of some commercial databases under heavy load
- Professional support depends on third-party providers, unlike commercial databases with dedicated support

4. MySQL

MySQL best features

- Benefit from its open-source nature, supported by a large community
- Retrieve data fast with high performance and speed
- Experience flexibility in MySQL, supporting multiple programming languages for diverse application development

MySQL limitations

- Faces challenges in extremely large-scale applications
- Less suitable for complex transaction systems compared to other database management systems

5. SQLite
   - Server-less and file based
   - Light-weighted, open source, simple for beginner
   - Not supporting multi-user well

6. MongoDB

- None SQL DB, None Logic Table structure, document based
- Easy to wrap data, but quite complicated to learn

Our Choice is Sqlite for this Class for its simplicity

SQLite for beginner

Installation

Install SQLite on Windows

Step 1 − Go to SQLite download page, and download precompiled binaries from Windows section.
https://www.sqlite.org/download.html

Step 2 − Download sqlite-shell-win32-*.zip and sqlite-dll-win32-*.zip zipped files.

Step 3 − Create a folder C:\>sqlite and unzip above two zipped files in this folder, which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.

Step 4 − Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue sqlite3 command, which should display the following result.

Check

```
C:\>sqlite3

SQLite version 3.7.15.2 2013-01-09 11:53:05

Enter ".help" for instructions
```

```
Enter SQL statements terminated with a ";"
sqlite>
```

Install SQLite on Mac OS

Step 1 − Go to SQLite download page, and download sqlite-autoconf-*.tar.gz from source code section.
https://www.sqlite.org/download.html

Step 2 − Run the following command −

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install (take note, if got permission issue, then use sudo make install)
```

Check

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Syntax

SQL query based language which has

SELECT
UPDATE
INSERT
DELETE
CREATE TABLE
DROP TABLE

With other functional Key works like:
Where
AND
OR
Order by
Group by
Count()
Sum()
Join
Union
Having
Distinct()
etc.

Sqlite C++ API

C/C++ Interface APIs

Following are important C/C++ SQLite interface routines, which can suffice your requirement to work with SQLite database from your C/C++ program. If you are looking for a more sophisticated application, then you can look into SQLite official documentation.

Sr.No.	API & Description
1	**sqlite3_open(const char *filename, sqlite3 **ppDb)** This routine opens a connection to an SQLite database file and returns a database connection object to be used by other SQLite routines. If the *filename* argument is NULL or ':memory:', sqlite3_open() will create an in-memory database in RAM that lasts only for the duration of the session. If the filename is not NULL, sqlite3_open() attempts to open the database file by using its value. If no file by that name exists, sqlite3_open() will open a new database file by that name.
2	**sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *data, char **errmsg)** This routine provides a quick, easy way to execute SQL commands provided by sql argument which can consist of more than one SQL command. Here, the first argument *sqlite3* is an open database object, *sqlite_callback* is a call back for which *data* is the 1st argument and errmsg will be returned to capture any error raised by the routine. SQLite3_exec() routine parses and executes every command given in the **sql** argument until it reaches the end of the string or encounters an error.
3	**sqlite3_close(sqlite3*)** This routine closes a database connection previously opened by a call to sqlite3_open(). All prepared statements associated with the connection should be finalized prior to closing the connection. If any queries remain that have not been finalized, sqlite3_close() will return SQLITE_BUSY with the error message Unable to close due to unfinalized statements.

Part III: Continuation Progressing in C++

If you like to have a review from simple concept overall, try below link.
https://www.geeksforgeeks.org/c-plus-plus/?ref=lbp

For official reference of user manual and document
https://en.cppreference.com/w/

for video content, there are plenty tutorial on youtube. There is one channel I like to recommend if you like to deepen your understand of C++.
https://www.youtube.com/watch?v=18c3MTX0PK0&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb
The Cherno - C++ series (100 + video)