# MQF 633 C++ for Financial Engineering

# Lecture 9: C++ with Python and Datebase

### 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>

1. Create a python file, and import ctypes, and then load C/C++ function from shared library
2. Make function call from py into C/C++ function in library.

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");

}

Complile it using command of

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 funciton 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() funciton. 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 declaired 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.

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)

# 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:

# 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 “muttable” behavour in python ctypes, we need to allocate memory buffer, this is almost like allocate memory in heap in C++.

# 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, initialized 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 from python into 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

#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.

#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 resturn 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

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 structure

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))

### 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](https://clickup.com/blog/free-database-software/) 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

1. Oracle Database

* Simliar 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.

1. PostGreSQL

Pro:

* Leverage its open-source nature for cost-effective data management
* Utilize advanced data types and performance optimization features
* Customize 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

1. 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

1. SQLite

* Serverless and file based
* Lightweighted, open source, simple for beginer
* Not supporting multi-user well

1. MongoDB

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

#### Our Chocie is Sqlite for this Class for its simplicity

#### SQLite for beginer

Installation

Install SQLite on Windows

Step 1 − Go to [SQLite download page](https://www.sqlite.org/download.html" \t "/Users/david/Documents\\x/_blank), 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](https://www.sqlite.org/download.html" \t "/Users/david/Documents\\x/_blank), 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 laguange 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 Progresssing 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)