Ceda Python Bindings

David Barrett-Lennard

17 Oct 2016

## Global functions

Consider that the following global function has been defined in C++, and reflected using the $function language extension

// C++ code

namespace mymaths

{

$function int32 sum(int32 x, int32 y)

{

return x + y;

}

}

Then the C++ global function can be called directly from python like this

# python code

print 'sum(1,2) = ' + `cns.mymaths.sum(1,2)`

## Vectors

Consider the following code that returns or takes a C++ xvector<int32>

// C++ code

namespace myvector

{

$function xvector<int32> Create()

{

return xvector<int32>();

}

$function void Print(const xvector<int32>& L)

{

std::cout << L;

}

}

A C++ vector can be wrapped in a Python object, supporting all the usual Python methods such as append(), extend(), insert(), remove(), pop(), index(), count(), sort() and reverse(). Also the functions len(), min(), max() are supported as well as + for concatenation. For example

# python code

L = cns.myvector.Create()

cns.myvector.Print(L + [1,2,3])

In addition L[i] provides read and write access to the ith element, and L[i:j] provides access to the slice from i up to but not including j. It is even possible to assign a slice or use the 'del' operator on a slice.

## Classes/structs

The following example declares a simple reflected C++ struct with a reflected member variable. Note that the global function CreateX() returns an X by value.

// C++ code

namespace myclass

{

$struct X

{

$void Set(int v) { a = v; }

$int32 a;

};

$function X CreateX()

{

return X();

}

}

The reflected C++ struct is wrapped in a python object allowing for read and write access to its reflected member variables and the ability to call its reflected member functions. For example:-

# python code

x = cns.myclass.CreateX()

x.a = 100

x.Set(200)

print 'x.a = ' + `x.a`

## Interfaces

The following example declares a simple reflected C++ struct that implements an interface that inherits from IObject.

// C++ code

$interface Ix : IObject

{

int32 Getx() const;

void Setx(int32 x);

};

$struct X isa Ix

{

X() : m\_x(0) {}

int32 Getx() const { return m\_x; }

void Setx(int32 x) { m\_x = x; }

int32 m\_x;

};

$function X CreateX() { return X(); }

Interfaces can be wrapped in python objects, providing support for calling all the interface methods directly from Python. For example:-

# python code

x = cns.CreateX()

x.Ix.Setx(10)

print 'x.Ix.Getx() = ' + `x.Ix.Getx()`

Note that the interface pointer is accessed using x.Ix, ie as though it were an attribute of the object that implements the interface.

## Exceptions

It is permissible for a reflected C++ function called from Python to throw an exception, as long as the exception inherits from IException.

For example

// C++ code

struct Apollo13Failure : public IException

{

virtual void Write(std::ostream& os) const

{

os << "Huston, we have a problem";

}

};

$function void FlyToMoon()

{

throw Apollo13Failure();

}

In the code below Python calls the function FlyToMoon() which throws an exception of type Apollo13Failure. As required, this inherits from IException. In Python, this causes a RuntimeError exception to be thrown, with a string value equal to the string written by the virtual Write() method of the C++ IException.

# python code

try:

cns.FlyToMoon()

except RuntimeError, e:

print e

## Summary

In addition to the examples above the following are supported:-

* Marshaling of pointers
* Boxing of signed and unsigned 64 bit integers
* Wrapping of functors
* Support for casting between different supported interfaces
* Read/write access to reflected global variables

These features are documented in the exPython samples.