Why And How To Add Scripting

Jason Turner

- http://github.com/lefticus/presentations
- http://cppcast.com
- http://chaiscript.com
- http://cppbestpractices.com
- C++ Weekly - YouTube
- @lefticus
- Independent Contractor

I prefer an interactive session - please ask questions

Who Is Currently Using Scripting?

My C++ Scripting Background

- First embedded script engine in C++ for a distributed Command and Control network, using Lua via SWIG ~2006
- Created SWIG Starter Kit November 2008 (current unmaintained)
- Started work on ChaiScript May 2009
- Consulted on C++/Scripting projects since 2010
- Contributed to SWIG Node/V8 binding generator

My C++ Scripting Background

I've worked on large C++ projects that are fully exposed to scripting

- 2209 classes/template instantiations, 66903 methods/functions exposed
- Supporting Ruby, Python, JavaScript, C#, Java all via SWIG
- Work equally well across Windows, Linux, MacOS

Why Do You Want Scripting?

I meet two kinds of C++ developers:

- Those who are already using scripting
- Those who have no idea why one would want scripting

Today we are going to focus on calling script from your C++, not calling C++ from your script

Why Do You Want Scripting?

Config Files

- Any application of any real complexity is going to need runtime configuration
- Often this ends up being a very simple, easy to parse file

Why Do You Want Scripting?

Config Files

Homebrew INI file

```
widget_1_x = 5
widget_1_y = 5
widget_1_width = 10
widget_1_height = 10
widget_1_name = "widget1"

widget_2_x = 15
widget_2_y = 5
widget_2_width = 10
widget_2_height = 10
widget_2_name = "widget2"
```

Why Do You Want Scripting?

Config Files

When what you really mean is

```
widget_count = 2

widget_1_x = 5
widget_1_y = 5
widget_1_width = 10
widget_1_height = 10
widget_1_name = "widget1"

widget_2_x = widget_1_x + widget_1_width
widget_2_y = widget_1_y
widget_2_width = 10
widget_2_height = 10
widget_2_name = "widget2"
```

- This holds true even if we are using something like JSON, XML, YAML

Why Do You Want Scripting?

Config Files

Scripted Config File

```
widget1 = Widget(5, 5,
                 10, 10, "widget1");

widget2 = Widget(widget1.x + widget1.width, widget1.y,
                 10, 10, "widget2");

add_widget(widget1);
add_widget(widget2);
```

Why Do You Want Scripting?

Config Files

By using a scripting engine we:

- gain flexibility
- save the effort of writing a parser
- can express our C++ types in our config files

Why Do You Want Scripting?

Application Logic

- By scripting application logic you can get much faster cycles for tweaking logic without recompiling
- You can use scripted application logic as a prototype, then convert to C++ when performance becomes an issue

Why Do You Want Scripting?

User Extensibility

- Common in javascript based applications
- github's atom editor
- etc

Our C++ applications can have the same level of flexibility and extensibility

Why Do You Want Scripting?

Runtime Configuration

Scripting can provide an easy way to read / change runtime parameters of a system.

Why Do You Want Scripting?

Other Ideas?

Languages Designed For Embedding

- Lua
- ChaiScript
- V8
- Qt Script
- Angelscript
- etc

Scripting Languages That Can Be Embedded

- Ruby
- Python
- etc

Tools We'll Cover

- SWIG: Simplified Wrapper and Interferface Generator
- Boost.Python: Python bindings interface layer provided by Boost
- sol2: Modern C++ bindings for Lua
- ChaiScript: Embedded scripting engine designed for C++

SWIG

- Parses C++ and generates bindings for various languages
- Last release: 2015-12-31 - in active development
- Wide range of compiler support

SWIG

Extensive Language Support

```
Allegro CL              C#
CFFI                    CLISP
Chicken                 D
Go                      Guile
Java                    Javascript
Lua                     Modula-3
Mzscheme                OCAML
Octave                  Perl
PHP                     Python
R                       Ruby
Scilab                  Tcl
UFFI
```

SWIG

Advantages

- Mostly automated, you don't have to specify your own interface (but can choose to)
- The generator can automatically create 'directors' to allow you to inherit from C++ classes in your script
- Can be configured to marshall exceptions between target language and C++

Disadvantages

- Multiple build steps with a code generator
- SWIG adds its own layer of indirection to handle overloads, which adds overhead
- Marshalling of exceptions can add a lot of generated code
- Sometimes SWIG can be very sensitive to type definition ordering

SWIG - Usage

1. Specify C++ interface you want exposed to your scripting language
2. Execute SWIG which generates a wrapper file
3. Compile generated SWIG output file
4. Initialize embedded scripting engine and load SWIG generated module
5. Execute script

SWIG/Ruby - C++ Interface

```
#ifndef EXPOSED_CODE_HPP
#define EXPOSED_CODE_HPP

#include <string>

std::string hello( const std::string & input );

#endif
```

```
#include "exposed_code.hpp"

std::string hello( const std::string & input ) {
  return "hello " + input;
}
```

SWIG/Ruby - SWIG .i Interface

```
%module EmbeddedScripting

%include <std_string.i>
%include "exposed_code.hpp"

%{
#include "exposed_code.hpp"
%}
```

SWIG/Ruby - C++ Embedding

```
extern "C" {
  // needed to provide the signature for initing our own module
  // this needs to match the signature of the module generated by SWIG
  void Init_EmbeddedScripting(void);
}

int main(int argc, char *argv[]) {

  // ruby initialization for embedding is completely undocument from what we could fin
  // this code is based on reading the source for the official irb
  ruby_sysinit(&argc, &argv);
  {
    RUBY_INIT_STACK;
    ruby_init();
  }

  Init_EmbeddedScripting();

  // This function defined elsewhere, available on github
  evalString(R"ruby(1000000.times { puts(EmbeddedScripting::hello('world')) })ruby");
  // Let's make this form of a raw string literal standard for syntax highlighting!
}
```

SWIG/Ruby - Compiling With CMake

```
add_custom_command(
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/EmbeddedScriptingRUBY_wrap.cxx"
  COMMAND "${SWIG_EXECUTABLE}"
          "-ruby"
          "-c++"
          -o "${CMAKE_CURRENT_BINARY_DIR}/EmbeddedScriptingRUBY_wrap.cxx"
          "${CMAKE_CURRENT_SOURCE_DIR}/EmbeddedScripting.i"
  DEPENDS "${CMAKE_CURRENT_SOURCE_DIR}/EmbeddedScripting.i"
          "exposed_code.hpp"
)

add_executable(EmbeddedScripting
 main.cpp
 exposed_code.hpp
 exposed_code.cpp
  "${CMAKE_CURRENT_BINARY_DIR}/EmbeddedScriptingRUBY_wrap.cxx"
 )

target_link_libraries(EmbeddedScripting ${RUBY_LIBRARY}  "dl" "crypt")
```

SWIG/Ruby - Generated File

```
SWIGINTERN VALUE
_wrap_hello(int argc, VALUE *argv, VALUE self) {
  std::string *arg1 = 0 ;
  int res1 = SWIG_OLDOBJ ;
  std::string result;
  VALUE vresult = Qnil;

  if ((argc < 1) || (argc > 1)) {
    rb_raise(rb_eArgError, "wrong # of arguments(%d for 1)",argc); SWIG_fail;
  }
  {
    std::string *ptr = (std::string *)0;
    res1 = SWIG_AsPtr_std_string(argv[0], &ptr);
    if (!SWIG_IsOK(res1)) {
      SWIG_exception_fail(SWIG_ArgError(res1), Ruby_Format_TypeError( "", "std::string
    }
    if (!ptr) {
      SWIG_exception_fail(SWIG_ValueError, Ruby_Format_TypeError("invalid null referen
    }
    arg1 = ptr;
  }
  result = hello((std::string const &)*arg1);
  vresult = SWIG_From_std_string(static_cast< std::string >(result));
  if (SWIG_IsNewObj(res1)) delete arg1;
  return vresult;
fail:
  if (SWIG_IsNewObj(res1)) delete arg1;
  return Qnil;
}
```

SWIG/Ruby - Generated File

```
SWIGEXPORT void Init_EmbeddedScripting(void) {
  size_t i;

  SWIG_InitRuntime();
  mEmbeddedScripting = rb_define_module("EmbeddedScripting");

  SWIG_InitializeModule(0);
  for (i = 0; i < swig_module.size; i++) {
    SWIG_define_class(swig_module.types[i]);
  }

  SWIG_RubyInitializeTrackings();
  rb_define_module_function(mEmbeddedScripting, "hello", VALUEFUNC(_wrap_hello), -1);
}
```

- Plus an additional 2000 lines of boilerplate code.
- If something goes wrong in here it can be difficult to debug why
- *However* this does amazing things, like handling dependencies and type info across multiple dynamically loaded modules

Boost.Python

Boost.Python

- Provides a wrapper layer for Boost <-> python
- Last significant update: 2009-11-17 (AKA boost 1.41.0) according to boost release notes
- Supports the compilers that Boost supports
- Why didn't we use pybind11? Learned about it late into preparing this talk and couldn't find examples on how to embed (instead of create module).

Boost.Python

Advantages

- Simple build process
- Easy to use interface

Disadvantages

- Must specify each thing you want bound to Python, no generator
- Not actively maintained

Boost.Python - Usage

1. Bind C++ functions to Python functions
2. Initialize embedded scripting engine
3. Load internally created module
4. Execute script

Boost.Python - Module Interface

```
#include <boost/python.hpp>

std::string hello( const std::string & input ) {
  return "hello " + input;
}

BOOST_PYTHON_MODULE(CppMod) {
  boost::python::def("hello", &hello);
}
```

Boost.Python - C++ Embedding

```
int main() {
  try {
    PyImport_AppendInittab( "CppMod", &initCppMod );
    Py_Initialize();

    boost::python::object main_module((
          boost::python::handle<>(boost::python::borrowed(PyImport_AddModule("__main__

    boost::python::object main_namespace = main_module.attr("__dict__");
    boost::python::object cpp_module( (boost::python::handle<>(PyImport_ImportModule("

    main_namespace["CppMod"] = cpp_module;

    boost::python::handle<> ignored(( PyRun_String(
R"python(
for number in range(1000000):
    print(CppMod.hello(\"world\"))
)python",
             Py_file_input,
             main_namespace.ptr(),
             main_namespace.ptr() ) ));
  } catch ( const boost::python::error_already_set & ) {
    PyErr_Print();
  }
}
```

Boost.Python - Compiling

```
g++ boost_python.cpp -I /usr/include/python2.7/ -lboost_python -lpython2.7
```

sol2

sol2

- Provides a wrapper layer between lua<->c++
- Last release 2016-05-03 - actively developed
- Supports Visual Studio 2015, Clang 3.5, G++ 4.9

sol2

Advantages

- Simple build process
- Easy to use interface
- Natural interaction with C++

Disadvantages

- Must specify each thing you want bound to Lua, no generator

sol2 - Usage

1. Create lua state object
2. Register C++ objects
3. Execute script

sol2 - C++ Embedding

```cpp
#include <sol.hpp>
#include <cassert>
#include <iostream>

std::string hello( const std::string & input ) {
  return "hello " + input;
}

int main() {
  sol::state lua;
  lua.open_libraries(sol::lib::base);

  lua.set_function("hello", hello);
  lua.script(R"lua(
    for i = 0,1000000,1  do print(hello("world")) end
)lua");
}
```

sol2 - Compiling

```
g++ ./sol2.cpp -I sol2/ -std=c++11 -I /usr/include/lua5.3/ -llua5.3
```

ChaiScript

ChaiScript

- Embedded scripting language co-designed by me specifically for C++
- Supports Visual Studio 2013, clang 3.4, g++ 4.5 (but this is changing as we move to C++14)
- Last release 2016-04-31 - actively developed

ChaiScript

Advantages

- Header only - no external deps
- Designed for integration with C++
- All types are the same and directly shared between script and C++ (double, std::string, std::function, etc)

Disadvantages

- Header only - compile times seem slow (but realisically probably not impact a real project much)

ChaiScript - Usage

1. Create ChaiScript engine object
2. Register C++ objects
3. Execute script

ChaiScript - C++ Embedding

```
#include <chaiscript/chaiscript.hpp>
#include <chaiscript/chaiscript_stdlib.hpp>

std::string hello( const std::string & input ) {
  return "hello " + input;
}

int main()
{
  chaiscript::ChaiScript chai(chaiscript::Std_Lib::library());

  chai.add(chaiscript::fun(&hello), "hello");
  chai.eval(R"chaiscript(for(var i = 0; i < 1000000; ++i) { print(hello("world")); } )
}
```

ChaiScript - Compiling

```
g++ ChaiScript.cpp  -ldl -pthread  -I ../../../ChaiScript/include/ -std=c++11
```

Conclusions

- I don't recommend embedding either Ruby or Python
- `PyErr_Print();`
- `PyErr_Print(); // global state`
- Global state means multithreading is somewhere between very difficult and impossible
- But there might be institutional reasons why either makes sense
  - Existing code bases
  - Existing knowledge bases
- Just because Ruby isn't recommended doesn't mean SWIG is not
  - SWIG / Lua, SWIG / V8 are good options

Questions?

Jason Turner

- http://github.com/lefticus/presentations
- http://cppcast.com
- http://chaiscript.com
- http://cppbestpractices.com
- C++ Weekly
- @lefticus
- Independent Contractor
- Stickers!