John Bandela, MD

CppComponents: A Modern, Portable C++ Component System

CPPCOMPONENTS

Why?

Why - Header Only Library Popularity

3 Header-Only Libraries

The first thing many people want to know is, "how do I build Boost?" The good news is that often, there's nothing to build.

Nothing to Build?

Most Boost libraries are **header-only**: they consist *entirely of header files* containing templates and inline functions, and require no separately-compiled library binaries or special treatment when linking.

Why? – Build Systems

Build Systems Used by C++ Projects
CMake
Boost Build
SCons
Gyp
Autotools/Make
MSBuild
Qmake/QBS
Others…

Why? – One C++

- Herb Sutter Talked about the Portable C++ Library Project
- Goal Slide from Presentation

Portable C++ Library (PCL)

GN'12

- Goals:
  - **Large** set of **useful** and **current** libraries.
  - Available on **all major platforms**.
  - **Shipped with** and **supported by** C++ implementations.
  - And **composable**, using consistent types.

  - Minimum: De facto availability as part of all major compiler products.
  - Ideal: De jure inclusion in Standard C++.

Reality

<regex>

Why? – Package managers

- Many languages such as Python, node JS, ruby, perl have package managers
- Package managers can greatly simplify discovering, installing and using libraries
- However, providing a precompiled binaries for every platform/compiler/standard library/debug vs release build is infeasible
- By providing a stable ABI, components allows a precompiled binary per platform with is much more feasible

Why? – Plugins and extensions

- If you want people to be able to write plugins for your application, you either need to create a bunch of extern C functions, or else are tied to a single compiler/standard library
- With a C++ component system, you can much more easily expose C++ classes and functions while still allowing others to use the compiler of their choice for plugins

Why – Fragile Base Class ABI

- Even when a single compiler and standard library is used, it is very easy to break ABI compatibility
- http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++

TransUnion

CPPCOMPONENTS

Introducing CppComponents

CppComponents

- https://github.com/jbandela/cppcomponents
- Boost License
- Header-only
- Tested on Windows with g++ 4.8.2/MSVC 2013
- Tested on Linux with g++ 4.8.2 / clang 3.4
- Not tested on Mac due to not having Mac, but should be relatively simple to port

CppComponents Demo

- See example1, example2, example3 - https://github.com/jbandela/cppcomponents_cppnow_examples

So how does this work

- See last year's talk on binary cross-compiler compatible interfaces
- CppComponents builds on extends last years talk
- Borrows many ideas and terminology from COM and WinRT. Though the implementation does not use any COM or WinRT, and uses only* C++11 to be portable

TransUnion

Non-standard but commonly implemented assumptions

- Can specify packing to generate create identical binary layout of a trivial structure across the two compilers
- Able to specify the platform calling convention for a static member function –see https://isocpp.org/wiki/faq/pointers-to-members

Review of the interfaces from last year

```
namespace detail{
    // Calling convention defined in platform specific header
    typedef  void(CROSS_CALL_CALLING_CONVENTION *ptr_fun_void_t)();
}

struct  portable_base{
    detail::ptr_fun_void_t* vfptr;
};
// base class for vtable_n
    struct vtable_n_base:public portable_base{
        void** pdata;
        …
};
```

TransUnion

```
// Our "vtable" definition
    template<int N>
    struct vtable_n:public vtable_n_base
    {
    protected:
        detail::ptr_fun_void_t table_n[N];
        void* data[N];
        enum {sz = N};
        vtable_n():vtable_n_base(data),table_n(),data(){
            vfptr = &table_n[0];
        }

    public:
        portable_base* get_portable_base(){return this;}
        const portable_base* get_portable_base()const{return this;}

    };
```

The cross_compiler interface

```
template<class T>
struct Interface
    :public cross_compiler_interface::define_interface<T>
{
    cross_function<Interface,0,std::string()> GetName;

    cross_function<Interface,1,void(std::string)> SetName;

 Interface()
        :SetName(this),GetName(this)
    {}
};
```

Vtable_caller and friends

- See snippets https://github.com/jbandela/cppcomponents_cppnow_examples

Supported parameter/return types

- (unsigned) char, wchar_t, char16_t, char32_t
- (u)int8/16/32/64_t
- float, double
- all (const) * and (const) & of the above
- (const) void*, bool
- std::basic_string,vector,pair,tuple,chrono::time_point, chrono_duration
- cppcomponents::string_ref(an adaptation of the boost version,with modifications to be able to tell if string is null-terminated)
- cppcomponents::use, cppcomponents::function

Define_interface

- ```
  template<
      class TUUID,
      class Base = InterfaceUnknown >
  struct define_interface
  ```
- Base class for cppcomponent interface with given uuid and a base interface
  ```

InterfaceUnknown

- Provides QueryInterface, AddRef, and Release
- Binary compatible with COM

uuid

```
template <
    std::uint32_t g1, // 8
    std::uint16_t g2, // 4
    std::uint16_t g3, // 4
    std::uint16_t g4, // 4
    std::uint64_t g5 // 12
>
struct uuid
```

TransUnion®

CPPCOMPONENTS_CONSTRUCT

- `#define CPPCOMPONENTS_CONSTRUCT(T,...)`

use

- `template<class Iface>`

  `struct use`
- Provides the ability to call interface functions
- Manages reference counting

Runtime class

- `template < const char* (*pfun_runtime_class_name)(),class... I  >`
  `struct runtime_class`
- Assembles various interfaces into a coherent whole

use_runtime_class

- ```
  template<class RC>
  using use_runtime_class = …;
  ```
- Inherits from each of the object interfaces
- Maps static functions to calls to static_interfaces
- Maps constructs to calls to factory_interface
  ```

implement_runtime_class

- ```
  template<class Derived, class RC>
   using implement_runtime_class = …
  ```
- Implements the interfaces of the runtime_class
- Provides InterfaceUnknown – QueryInterface, AddRef, Release – Implementation
- Maps object interfaces to member functions
- Maps factory interfaces to constructors
- Maps static interfaces to static functions
- Note: after the first interface, the object and static interface functions are mapped to Interface_Function
  ```

implement_runtime_class

- Has a static variable of a class that implements the factory and static interfaces
- The factory and static interface class's constructor registers the instance in a module local factory map

CPPCOMPONENTS_REGISTER

- ```
  #define CPPCOMPONENTS_REGISTER(T) namespace{auto
  CROSS_COMPILER_INTERFACE_CAT(cppcomponents_registration_variable
  , __LINE__) = T::cppcomponents_register_fsi(); void
  CROSS_COMPILER_INTERFACE_CAT(dummyfunction,
  CROSS_COMPILER_INTERFACE_CAT(cppcomponents_registration_variable
  , __LINE__))
  (){(void)CROSS_COMPILER_INTERFACE_CAT(cppcomponents_registration_
  variable , __LINE__)  ;} }
  ```
- Makes sure the static variable of implement_interface gets instantiated
  ```

CPPCOMPONENTS_DEFINE_FACTORY

- See snippet - https://github.com/jbandela/cppcomponents_cppnow_examples
- Defines the only exported functions for the dynamic libraries

Constructing a use_runtime_class

- Ask for the activation factory for our runtime class id – getting back use<InterfaceUnknown>
- QueryInterface<FactoryInterface>()
- Based on what types we were passed in, call the appropriate factory interface function

Getting the activation factory

- Look up the module that implements the runtime class id
- Load and initialize that module if necessary
- Ask the module to provide us the activation factory if it implements it
- The module will refer to its local factory map (in which the constructor of the factory static implementations registers themselves) and return the activation factory

Mapping from runtime class id to module name

```
• struct IStringFactoryCreator : public
  cppcomponents::define_interface<cppcomponents::uuid<0x33e78ea2,
  0xb89f, 0x479a, 0x8f10, 0xfd3b4234b446>>

{

void AddMapping(std::string class_name, std::string module_name);

use<InterfaceUnknown> GetClassFactory(std::string class_name);

use<InterfaceUnknown> GetClassFactoryFromModule(std::string
class_name, std::string module_name);

void FreeUnusedModules();


CPPCOMPONENTS_CONSTRUCT(IStringFactoryCreator, AddMapping,
GetClassFactory, GetClassFactoryFromModule, FreeUnusedModules)

};
```

TransUnion

Runtime class id conventions

- Runtime class id are of the form “<Module>!<Class Name>
- By default, will load Module.dll /.so and ask the module for the activation factory for the runtime class id
- If “<Module>!” Is absent will look if there is a prefix mapping. Note a prefix mapping will also override the default <Module>. If no prefix mapping, will look in the local factory map of whoever is providing IStringFactoryCreator
- If it is of the form “!<Class Name>” will only look in the local factory map

How modules are loaded

- Uses LoadLibrary (Windows) and dlopen(Linux)
- Calls the exported function cppcomponents_module_initialize passing in the IStringFactoryCreator from our main executable
- The module then sets that IStringFactoryCreator as the one to use to look up class id to module name mapping
- This assures us that changes made to class id to module mapping will be consistent across all dynamic libraries. This allows a very simple form of dependency injection

TransUnion

CPPCOMPONENTS

Beyond the Basics

Simple Dependency Injection

- Example5 - https://github.com/jbandela/cppcomponents_cppnow_examples

Interface Overloads and Templates

- The cross compiler interface can neither have overloads or template functions
- Sometimes it can be useful to have in the interface
- Use CPPCOMPONENTS_INTERFACE_EXTRAS with this->get_interface() for object interfaces and factory interface
- Use CPPCOMPONENTS_STATIC_INTERFACE_EXTRAS with Class:: for static interfaces
- Overload TemplatedConstructor in CPPCOMPONENTS_INTERFACE_EXTRAS in a factory interface to have a template constructor
- See snippet

Parameterized Interfaces

- Sometimes we want to parameterize an interface on a template parameter
- However, how do we need to guarantee that the uuid for each interface is unique
- Version 5 uuid’s use sha1 to generate a uuid
- combine_uuid combines multiple uuid using sha1 to generate a version5 uuid
- Specializing cppcomponents::uuid_of<> allows us to make sure each type has a uuid associated with it
- See future snippet for example

Dynamic loading

- Sometimes we want to be able to load a module explicitly
- This is very useful, for example, if you are working with plugins
- See example6 - https://github.com/jbandela/cppcomponents_cppnow_examples

Call by name

- Especially when interfacing with either configuration or scripting, it can be useful to call an interface function by name
- See example7 - https://github.com/jbandela/cppcomponents_cppnow_examples

CPPCOMPONENTS

Concurrency: Future, Promise, Channel

Demo

- Example4 - https://github.com/jbandela/cppcomponents_cppnow_examples

Discussion of Future, Promise, Channel

- See snippet - https://github.com/jbandela/cppcomponents_cppnow_examples

TransUnion®

CPPCOMPONENTS

Future Directions

Ongoing work

- A boost asio based implementation of executors and async network io, timers
- A boost coroutine based await implementation
- Wrapper for async use of libcurl
- Ccpm – A C++ Components Package Manager

Future Plans

- Port to Mac
- ? Remote components over network
- ? QML/Javascript interface
- ? COM/WinRT wrappers
- Http server library

Call to try it out

- Code is at https://github.com/jbandela/cppcomponents
- Try it out, give feedback
- Is there an existing library, you wish you could always conveniently use? Write a cppcomponents module for it. I am happy to help you if you run into any questions.
- Together, let's build a large C++ components ecosystem across various platforms.

Questions?