Boost.Extension
Boost.Reflection

Jeremy Pack
5 May 2008

The Original Motivation

- Moving object tracking.
- Real-time vs. Post-processing
- Low-speed, low resolution cameras or many high-speed cameras.
- Simple IR, standard color or highly detailed multi-spectral IR imagery.

Varying Needs of the Application

- Many different types of input and output.
- Many different scenarios in which it would be used.
- Intellectual property issues. Multiple customers would use this software, some with custom input and output formats, or highly specialized image formats.

Initial Approach

- Create a variety of builds. Give different builds to each customer, to match their needs.
  - What if a customer needs a change to the software in the future for their proprietary data type? This software could be used for years.
    - Should each client receive full source code in case they need additional capabilities in the future?
  - Difficult to separate the code out according to each client.

An Extensible Approach

- Only common functionality is placed into the main executable.
- Each customer is then given two sets of files:
  - Shared libraries for their specific data types, UI needs etc.
  - An SDK to develop plugins in the future to handle any new requirements for the software.

Shared Libraries

- Normally, shared libraries are loaded by an executable when it starts.
- Some Boost libraries support this. An executable can link to a shared library version of, for instance, Boost.Thread.
- Function and variable locations are then resolved at load time.
- Shared libraries can also be loaded at will, during program execution.
- They can then be searched for functions or data.

Problems With Shared Libraries

- Separate symbol tables are required in each shared library.
- RTTI merging: Some compilers do it. Some don't.
- Optimizations are harder because the code must be position independent.
- The following may break binary compatibility between an application and a shared library (among other things):
  - Any part of shared class changes. Variable ordering, public/private/protected, virtual function implementations, add/remove variables/functions etc.
  - Different compiler or compiler options used, or headers included differently.

Inefficiency of Shared Libraries

- Position Independent Code:
  - Since the shared library could be loaded to any part of the address space, it cannot hard code addresses of its functions and data. This must be resolved when the library is loaded.
  - They can take a really long time to load.
  - OS's attempt to mitigate this in various ways.
- Extra levels of indirection in non-virtual function calls.
  - No inlining across library boundaries.

Should Shared Libraries be Built-in?

- The interface is fairly uniform across operating systems:
  - Functions to load and close libraries, and find symbols.
  - Libraries can be loaded multiple times, and are reference counted.
- A major benefit that could be gained from building shared libraries into the language would be better type safety across shared libraries.

Shared Library Support as a Library

- Automatic loading of named modules:
  - shared_library m("my_module_name");
  - m.open();
  - m.get<int (float)>("function_name")(5.0f)
  - m.close();
- It's clear that this can be done as a header-only library well, instead of as built-in functionality.

Extension: Shared Library Internals

```
class shared_library
{
public:
  bool is_open(){return handle_ != 0;}
  bool open() {
    return (handle_ = load_shared_library(location_.c_str())) != 0;
  }
  bool close() {
    return close_shared_library(handle_);
  }
  shared_library(const std::string& location, bool auto_close = false)
    : location_(location), handle_(0), auto_close_(auto_close) {
  }
#define BOOST_PP_ITERATION_LIMITS (0, \
    BOOST_PP_INC(BOOST_EXTENSION_MAX_FUNCTOR_PARAMS) - 1)
#define BOOST_PP_FILENAME_1 <boost/extension/impl/shared_library.hpp>
#include BOOST_PP_ITERATE()
protected:
  std::string location_;
  library_handle handle_;
  bool auto_close_;
};
```

Boost.PreProcessor

- Boost.PreProcessor macros are used to define multiple parameter versions of templated functions. This is user-configurable, but the default number of function parameters is 10.
- Thanks to various bits of help from the Boost community, the functions using PreProcessor macros are relatively easy to use in a debugger.

Shared Library Type Safety - Name Mangling

- comp.lang.c++ 1996 - Varied rants about the fact that there is neither a standardized mangling format, nor a linker that supports tracking names by their type, calling convention, etc. without mangling.
- 2008 - not much has changed.
- Symbols in shared libraries must be declared ‘extern “C”’ to avoid mangling, and thus be visible to users of the library.
- This causes type information about the function or data to be lost. Boost.Extension must cast a void pointer back to the user requested type.
- To minimize this problem, preferred usage of shared_library is to declare only one “extern” entry point, and pass back any necessary reflections, factories or data there.

Shared Library Type Safety - Compilation Options

- If an application attempts to load a shared library that was compiled differently, in some way that breaks ABI compatibility, serious issues can occur.
- To avoid this, Boost.Extension (or the user) could put data into each library defining a number of the parameters with which it was compiled - ie:
  - Compiler and OS
  - Version of Boost
  - 32/64 bit
  - etc.

If Plugins (using factories) were built into C++

- Query a shared_library (or the main executable) for all classes that are derived from a given class. Return factories for those classes.
- Would a class have to be declared to be externally visible? What about template classes?
- What if we need to include arbitrary information about a plugin? Versioning? Dependency lists?
- What if we can’t modify the class itself?
- What if we want arbitrary type information?

Plugins as a Library

- Call a function from a shared library (or the current module) that returns a list of factories.
- Index them by type information, versioning, constructor types or whatever is appropriate by application.

Declare factories in your shared library

```
class world : public word {
public:
  virtual const char * get_val(){return "world!";}
};
class hello : public word {
public:
  virtual const char * get_val(){return "hello";}
};
extern "C"
void BOOST_EXTENSION_EXPORT_DECL
extension_export_word(factory_map& fm) {
  fm.get<word, std::string>()["hello class"].set<hello>();
  fm.get<word, std::string>()["world class"].set<world>();
}
```

Access factories from your executable

```
// Create the factory_map object - it will hold all of the available
// constructors. Multiple factory_maps can be constructed.
factory_map fm;
load_single_library(fm, "libHelloWorldLib.extension",
                    "extension_export_word");  // This is a convenience function
// Get a reference to the list of constructors for words.
std::map<int, factory<word> > & factory_list = fm.get<word, int>();
for (std::map<int, factory<word> >::iterator current_word =
      factory_list.begin(); current_word != factory_list.end();
    ++current_word) {
  std::auto_ptr<word> word_ptr(current_word->second.create());
  std::cout << word_ptr->get_val() << " ";
}
```

Boost.Extension Factory Support

- Factories are indexed by type, constructor signature, and a templated Info field.
- One could look up all factories for objects of type MyObject with a constructor that accepts a given set of parameters.
- Each factory is declared by the programmer.
- Factories can be in shared libraries or in the current module.

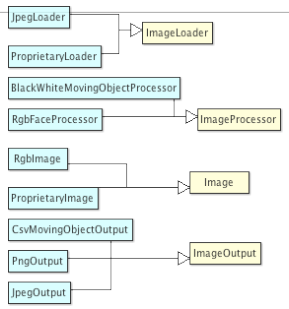
Compatibility: Boost.Function

- It is still an open issue whether or not to use Boost.Function internally.
- In general, types are used that are compatible with Boost.Function - function pointers, standard function objects, etc.
- Boost.Functional/Factory objects - like most function objects - can be passed back by shared library functions in a similar manner.

Building our Image Processing Application

- This is a sample being worked on for the Extension examples/ folder.
- The GUI is HTML. A slightly modified version of one of the Asio HTTP servers is used.
- Any volunteers to create a simple OpenGL based C++ GUI library?

An Initial Class Structure



Many of these classes have interdependencies: Perhaps an RgbFaceProcessor needs a reference to an ImageLoader for color images?

How to add a new class

- Build a shared library including the class.
- Add an externally visible function to export the function for the class.
- Install it with any shared libraries or configuration files that it depends on.
- Let the application know that this new library exists.
  - This can be automatic, based on a config file, etc.

If Reflection Were Built Into C++

- Java-style reflection would have high overhead in C++. How much reflection information would have to be generated for code using template metaprogramming?
- Reflect only certain classes, ie:
  - reflected class MyClass {};
- Shared libraries share reflection info when loaded.
- RTTI-based. One could iterate through member functions, constructors or data members and retrieve the typeid() values, as well as names.
- Should private functions be reflected? Protected? Data members?

The Macro Approach

Some implement reflection through heavy use of macros. Sample reflected class:

```
REFLECTED_CLASS(myClass, myBase) {
  REFLECTED_METHOD(myMethod, public) {
  }
  REFLECTED_METHOD(myOtherMethod, private) {
  }
};
```

Another Approach: Parsing Source Files

- A separate build step parses each desired .h file and generates type information about the class.
- This is hard: your parser has to know a lot of things that the compiler knows. It has to correctly handle all macros in the file, all templates, all namespaces...
- Some tools can handle this - such as gccxml.
- This isn't acceptable for Boost - we need a cross-platform solution.

Parsing Debug Information

- Platform-specific.
- What if we want a release build?
- Virtual methods?
- const? mutable? private? protected?

Other Notes on Current Methods

- Usually, only default constructors are supported.
- They often require that all variables be wrapped in some sort of base Object.
- The most powerful of them are very compiler specific.

Boost.Reflection

- The basic interface is similar to Boost.Python.
- Designed to be used in the same way as factories in Boost.Extension.
- Reuses much of the Boost.Extension code and techniques.

Sample Reflection code - declaration

```
class suv {
public:
  suv(const char * name);
  const char* get_type(void);
  void start();
  bool start(int target_speed);
  short year;
};

reflection r;
r.reflect<suv>()
    .constructor<const char*>()
    .function(&suv::get_type, "get_type")
    .data(&suv::year, "year")
    .function<void>(&suv::start, "start");
    .function<bool, int>(&suv::start, "start");
```

Sample Reflection code - usage

```
std::map<std::string, reflection> reflection_map;
shared_library lib("libcar_lib.extension");
lib.open();
lib.get<void, std::map<std::string, reflection> &>
  ("extension_export_car")(reflection_map);

 reflection& reflection =
  reflection_map["suv"];
instance_constructor<const char *> first_constructor =
  first_reflection.get_constructor<const char *>();

instance first_instance = first_constructor("First Instance");
boost::reflections::function<const char *> type_function =
  first_reflection.get_function<const char *>("get_type");
std::cout << "First reflection type: " << type_function(first_instance)
        << std::endl;
```

Boost.Reflection Features

- No macros required of users - only templates. Macros are only used internally (Boost.PreProcessor code for handling function signatures).
- Non-intrusive.
- Allows RTTI - or other type info mechanisms.
- Provides support for dependency tracking between reflected classes.
- Functions (including constructors) can be called without knowing their signature. (The interface for this functionality has not been finalized)
- Iterate through functions and data of a reflected class.
- Reflect non-member functions that operate on the reflected type - not finalized.

Overhead of Boost.Reflection

- Finding a function or a constructor requires a lookup.
- Once a function is found, the overhead of calling the function is that of calling one function pointer, and then calling a member function pointer.
  - It would be possible to decrease this overhead for many functions, but certain types of functions pose problems:
    - Virtual functions.
    - Functions defined by base classes.
  - This is the same overhead in Boost.Function.

Using Reflection in our Image Processing Application

- Our standard objects - ImageLoaders, some of the algorithms, image output etc. can be handled pretty well by factories.
- In other cases, we may want more generalized abilities.
- Reflected objects can each represent different views on the data as a whole. Each can require that certain other objects be loaded before they can be created, by adding references to them as parameters in the constructor.
- Instead of trying to fit every object into the class hierarchy already defined, we can define any object that seems to be needed by the ever-changing requirements.
- If done carefully, this can lead to well-defined, minimal dependencies between objects in the system.

Other Obvious Applications

- IDEs or other applications where very different functionality is needed by different users.
- Applications with a variety of input or output types.
- Data applications where different views of the data can be useful.
- A very large application where usually only small isolated parts are changed at a time.
- Applications that need to keep running when adding new functionality.
- Audio or Video apps - changing codecs, etc.

Interface Choices

- Use Boost.Function internally?
- Mimic Boost.Python completely? Make them interoperable?
- Use member function syntax for reflected methods?
- Have instances reference the correct reflection?
- Use Boost.Function style preferred/portable syntax? (easier in some places than others)

Type Information

- Type information comparison has to be handled differently depending on the platform and compiler, depending on how the RTTI is shared between modules:
  - &type1 < &type2 // Fast.
  - type1 < type2 // Sometimes fast.
  - strcmp(type1.name(), type2.name) // never fast
  - strcmp(type1.raw_name(), type2.raw_name()) // Some MS compilers, this is faster than name()

Avoiding Shared Library Inefficiencies

- Boost.Reflection could be implemented with an almost identical interface as running across processes, using shared memory or queues for method calls, and by modifying types in transit.
  - Special types would be needed, for example, for references: reference<int>
  - Reflectable parameters would have to be defined, and the method for serialization also defined.
  - This would be most appropriate for simple functions - passing integers, strings, arrays etc.

Distributed Reflection?

- By building a large, fault tolerant, probably Boost.Asio-based framework, it would be possible to reflect across networks.
- The object would be created and hosted in one process, and interacted with in another.

Other Possible Future Work

- Generate C++ code and load it - C++ interpreter? Also allow interaction with previously coded objects?
  - C++ compilation can be slow - especially if there are a lot of headers (see Boost.org).
- Load a reflection for a C++ class after being given the source code?

Questions?

- See http://boost-extension.blogspot.com for more details, and links to current documentation.

Other Contributors

- Mariano Consoni
- Hartmut Kaiser
- Maik Beckman
- Janek Kozicki
- Everyone else who has reported bugs or suggested changes!

References

- http://www.iecc.com/linker/linker10.html
- http://www.dwheeler.com/program-library/Program-Library-HOWTO/x36.html
- http://www.garret.ru/~knizhnik/cppreflection/docs/reflect.html
- http://boost-extension.blogspot.com