Rich Pointers

Document Number: ISO/IEC JTC1 SC22 WG21 N3340=12-0030
Date: 2012-01-10
Authors: Dean Michael Berris <dberris@google.com>, Matt Austern <austern@google.com>, Lawrence Crowl <crowl@google.com>

Table of Contents

Overview

This paper proposes the definition of a standardized rich pointer and type descriptor construct in C++ to allow for standardized and efficient runtime introspection of types and objects. Here we propose the syntax and semantics for a rich pointer and examples of programming problems addressed by this construct.

Problem Statement

There are some application areas today where rich runtime information about objects is crucial. These application areas include:

- Highly available server applications that have to stay up as a long running process and enable for dynamic adaptation to changing requirements. Upgrades to these applications currently need to be closely coordinated and tightly controlled, usually requiring that the service be actually brought down when an upgrade is required. There are solutions that exist which involve dynamically loaded libraries but changing the design of already packaged types in the system usually require downtime to rebuild and relink the binary.
- Applications that embed dynamic programming languages typically settle for either a static interface to which the embedded dynamic language runtime, or a pure data interface implementing a (static) protocol between the embedded environment and the host application. Usually the dynamic types that can be generated in these embedded virtual machines are only usable in the context of these virtual machines limiting the interactions by which these embedded types interact with the host application.
- Distributed computing systems written in C++ are largely tied to static interfaces and types because of the limitation of the programming language. We currently do not have a standard programmatic way of dynamically generating types and referring to objects of these types and streaming objects of these types from one system to another. Current state of the art relies on code generators, domain specific languages, and even communication frameworks to achieve remote procedure calling and sharing state across elements in a distributed system.
- Applications use machine learning and dynamic modeling of environments through continuous refinement of data structures in memory rely on the capability to treat code as data, or at least be able to inspect the state and relationship between types in a hierarchy of types. The current limitations of the language force the applications that deal with constantly changing structures and types is to model them in runtime as merely data, losing much of the power of the programming language's runtime facility for efficiently modeling types and objects in the process.
- Graphical user interfaces typically rely on being able to inspect in-memory structures to represent graphical elements to be rendered on screen. Almost all the sufficiently advanced graphical user interface toolkits now use a statically-defined and very rigid type system and implement a runtime meta-type system because the programming language currently does not have a facility of doing proper rich runtime introspection of types.

There are several problems that this paper aims to address. Here are some of these questions:

- How do we find out what the type of an object is at a given memory location? Type erasure allows us to expose a generic API that works well across module boundaries,

but being able to preserve type information across these module boundaries without having to rebuild and relink binaries is also as powerful.

- How do we print the structure of a dynamic type at runtime? Currently there are no ways to do this dynamically without resorting to manual, static, and expensive checks on types that are members of a statically-known hierarchy of types. There exists no standard means of knowing the type of a given object in memory at runtime with the current features of the language, especially when referred to using either a `void *` or a base type pointer.
- How do we determine the relationship between any two given types at runtime? The current way of doing this requires manual checks for whether one type can be dynamically cast to another, and inferring from the result what the potential relationship between the types is. There is currently no way of determining what types a given type is derived from, what type of inheritance (public, private, protected, virtual, etc.) these relationships are, and whether a given type is an abstract class or whether it is a final class, etc.
- If we were able to create new types at runtime, how do we describe these types and inspect them? For applications that rely on live updates for high availability and remote procedure calling systems, being able to reconstitute a type dynamically based on data obtained externally via I/O is crucial. Currently the only way to allow this is to implement a runtime type system by hand and perform all type inspections by hand, unable to leverage the rich type system that C++ provides. This is also important in applications where an embedded just-in-time (JIT) compiler can create new structures programmatically as well as allowing types generated in embedded runtime environments to be exported to the host application.

Proposed Solution

The problems raised above point to general dynamic programming utilities called runtime introspection and reflection. Enabling introspection and reflection at runtime has traditionally been costly and almost always requires the concept of a virtual machine for it to be doable. In this paper we describe a mechanism for enabling introspection and potentially reflection without the need for a heavyweight virtual machine or runtime to make it possible. The benefits include but are not limited to: better runtime debugging and instrumentation for profiling as well as for first-class garbage collection support.

Rich Pointers

The smart pointer idiom has been generally accepted in practice largely for the utilities afforded to us by these smart pointers. The standard library now contains three smart pointers: `unique_ptr`, `shared_ptr`, and `weak_ptr`. These smart pointers encapsulate generic patterns for dealing with memory ownership and management and makes it transparent to users.

Following the smart pointer idiom this paper proposes a concept called rich pointers which not only carry the memory location of a given object but also a reference to an immutable representation of the type of this object (we call these type descriptors). The proposed syntax for a rich pointer follows the general style of a pointer but instead of using the * symbol to denote a pointer we use the percent "%" symbol to do so.

```
struct foo {
  foo() {}
  ~foo() {}
};

foo %p = new (std::rich) foo;
```

For the most part a rich pointer behaves like a normal pointer. Dereferencing will yield a reference to the object pointed to but you can no longer perform normal pointer arithmetic. When a rich pointer is assigned to a normal pointer, the memory location is transferred and the reference to the original type descriptor is dropped (this is to ensure backward compatibility).

What rich pointers support that would be hard to support with normal pointers is the notion of preserving type information even across casts even to `void%`. To illustrate more appropriately:

```
void %q = p;
foo %r = rich_cast<foo%>(q);
assert(type_descriptor(q) == type_descriptor(r));
```

Because p and q point to the same object in memory, getting the type descriptor of both pointers will yield the same type descriptor. In fact, any object of type `foo` in memory when referred to via a rich pointer will have the same type descriptor.

Type Descriptors

The reason we're proposing a language feature to support rich pointers as opposed to a smart pointer implemented in C++ is so that the generation of type descriptors can be implemented by the compiler (whether an embedded JIT compiler, or a normal compiler). Type descriptors are maintained by the language runtime are tied directly to the runtime type information (RTTI) implementation.

To access a type descriptor, we're introducing a standard function called `type_descriptor` which returns a pointer to an immutable object representing that unique type. The following types

encapsulate various kinds of type information:

```
struct function_descriptor_t {
  char const *name;  // null terminated function name
  type_descriptor_t const *result_type;
  type_descriptor_t const *args[];
  enum { NORMAL, VIRTUAL, CONST, VOLATILE,
         CONSTEXPR, STATIC, NAMESPACE }
    qualifiers_t;
  int qualifiers;  // binary OR'ed qualifiers_t values.
  enum { CONSTRUCTOR, DESTRUCTOR, FREE, MEMBER }
    function_type_t;
  function_type_t function_type;
  type_descriptor_t *member_of;  // pointer to enclosing type.
  unspecified_type callable;  // bounded callable function object.
};

struct field_descriptor_t {
  char const *name;  // null terminated member name
  type_descriptor_t const *type;
  enum { PRIVATE, PUBLIC, PROTECTED }
    access_qualifiers_t;
  access_qualifiers_t access_qualifier;
  enum { INSTANCE, STATIC }
    membership_qualifiers_t;
  membership_qualifiers_t membership_qualifier;
};

struct type_descriptor_t {
  char const * name;  // null terminated type name

  struct inheritance_descriptor_t {
    enum { PRIVATE, PUBLIC, PROTECTED }
      access_type_t;
    access_type_t access_type;
    enum { FINAL, VIRTUAL }
```

```
      inheritance_type_t;
    inheritance_type_t inheritance_type;
    type_descriptor_t const *base;
  };

  inheritance_descriptor_t *bases[];

  struct method_descriptor_t {
    enum { PRIVATE, PUBLIC, PROTECTED }
      access_type_t;
    access_type_t access_type;
    enum { NORMAL, FINAL, VIRTUAL }
      inheritance_type_t;
    inheritance_type_t inheritance_type;
    function_descriptor_t const *type;
  }

  function_descriptor_t *methods[];
  method_descriptor_t *members[];
  size_t size;
};
```

The prototypes for the type_descriptor function is given below:

```
template <class T>
type_descriptor_t const *
type_descriptor(T%);  // For rich pointer types.

template <class T>
type_descriptor_t const *
type_descriptor(T*);  // For normal pointer types.

template <class R, class T...>
function_descriptor_t const *
type_descriptor(R(T...) *);  // For function pointers.
```

What type descriptors contain is rich type information regarding a specific type. The idea is, all the types ever referred to via a rich pointer in a translation unit will expose a type descriptor. This type descriptor is then considered immutable and attempts to programmatically change an existing type descriptor invokes undefined behavior.

A compiler will be able to generate these type descriptors and make them available to the runtime implementation. As an example, the following complete translation unit should generate the also provided type descriptors:

```
#include <iostreams>
#include <string>
#include <runtime>

struct foo {
  foo() : a_(0), b_("") {}
  foo(foo const &) = delete;
  foo(foo &&) = delete;
  foo& operator=(foo) = delete;
  void bar() { /* do nothing */ };
  ~foo() { /* do nothing */ };
 protected:
  int a_;
 private:
  std::string b_;
};

int main(int argc, char *argv[]) {
  foo %f = new(std::rich) foo;
  // The following should print "::foo".
  std::cout << type_descriptor(f)->name << std::endl;
  delete f;
  return 0;
}

// Type descriptors, generated by compiler and registered through the
// runtime interface. Note that only types that are used through the
// rich pointer syntax will automatically have the type descriptors
```

```
// generated for them by the compiler along with the dependent types.
void __register_types() {
  // For convenience...
  typedef type_descriptor_t::method_descriptor_t method_descriptor_t;
  static function_descriptor_t const foo_void_bar {
    “::foo::bar”,  // function name.
    &void_type,  // result type, primitive types defined in <runtime>.
    {nullptr},  // no arguments.
    function_descriptor_t::NORMAL,  // qualifiers.
    function_descriptor_t::MEMBER,  // type.
    nullptr,  // member_of, nullptr at this time.
    std::bind(&foo::bar, _1),  // callable function, 1st arg is “this”.
  };
  static function_descriptor_t const foo_ctor_0 {
    “::foo::foo”,  // function name.
    nullptr,  // result type, nullptr for constructors.
    {nullptr},  // no arguments.
    function_descriptor_t::NORMAL,  // qualifiers.
    function_descriptor_t::CONSTRUCTOR,  // type.
    nullptr,  // member_of, nullptr at this time.
    &__constructor<foo>::callable,  // implementation defined.
  };
  static function_descriptor_t const foo_dtor {
    “::foo::~foo”,  // function name.
    nullptr,  // result type, nullptr for destructors.
    {nullptr},  // no arguments.
    function_descriptor_t::NORMAL,  // qualifiers.
    function_descriptor_t::DESTRUCTOR,  // type.
    nullptr,  // member_of, nullptr at this time.
    &__destructor<foo>::callable,  // implementation defined.
  };
  static type_descriptor_t const foo_type {
    “::foo”,  // type name.
    {nullptr},  // no bases.
    {
      { method_descriptor_t::PUBLIC,
```

```
      method_descriptor_t::NORMAL,
      &foo_ctor_0
    },
    { method_descriptor_t::PUBLIC,
      method_descriptor_t::NORMAL,
      &foo_void_bar
    },
    { method_descriptor_t::PUBLIC,
      method_descriptor_t::NORMAL,
      &foo_dtor
    },
    nullptr
  },  // functions.
  {
    { “a_”,  // member name.
      &int_type,  // member type, primitives defined in <runtime>.
      field_descriptor_t::PROTECTED,  // access qualifier.
      field_descriptor_t::INSTANCE  // membership qualifer.
    },
    { “b_”,  // member name.
      &string_type,  // member type, in compilation of <string>.
      field_descriptor_t::PRIVATE,  // access qualifier.
      field_descriptor_t::INSTANCE  // membership qualifier.
    }
  },
  sizeof(foo)  // the statically determined size.
};

// We then wire up the members.
const_cast<function_descriptor_t*>(&foo_ctor_0)->member_of
  = &foo_type;
const_cast<function_descriptor_t*>(&foo_void_bar)->member_of
  = &foo_type;
const_cast<function_descriptor_t*>(&foo_dtor)->member_of
  = &foo_type;
```

```
  // This part is the implementation defined part.
  std::__runtime_register_type_descriptor<foo>(&foo_type);
}
```

There are two ways of making the runtime manage descriptors: registration and invalidation.

When a type descriptor is registered in the runtime, it must not allow the redefinition of an existing descriptor tied to a specific type. Attempts to register a different descriptor to an already registered type should fail. This enables dynamically loaded shared libraries to add new type descriptors to an existing runtime environment.

Invalidating a type descriptor does two things: invalidates all rich pointers to objects of the invalidated type -- attempts to dereference the pointers of an invalidated type will then lead to exceptions that can be handled either by user code or by a special invalidation handler associated with the type. Invalidation handlers can be registered using the same mechanism by which new types are registered. This enables dynamically loaded shared libraries to invalidate types that it intends to either explicitly stop supporting or upgrade to a newer version.

The following functions are proposed to be added for registration and invalidation of types at runtime:

```
// register_type is meant to be called at module initialization time
// depending on how dynamic/shared or static modules are initialized.
// The result is a boolean indicating success and a pointer to the
// new immutable type descriptor. This will only return true if the
// type T is valid and that it has not been previously registered.
//
// Example usage:
//     bool ok;
//     type_descriptor_t const *p;
//     tie(ok, p) = register_type(new(std::rich) foo);
//     assert(ok);
template <class T>
tuple<bool, type_descriptor_t const *>
register_type(T%);

// Returns whether the type of the pointer is invalidated.
bool type_invalidated(void%);
```

```
// invalidate_type is meant to be called at module initialization time
// depending on how dynamic/shared or static modules are initialized.
// The result contains a boolean indicating success and pointers to the
// invalidated type descriptor and new type descriptor. It is important
// that this is called at module initialization time when the current
// runtime context and module-specific runtime contexts are available.
// Calls to new(std::rich) within module scope will always use the
// module-specific runtime context. During module initialization, calls
// to type_descriptor(...) will always refer to the current runtime
// context. If a type is not already registered through
// register_type(...) then calling invalidate_type will return the tuple
// (false, nullptr, descriptor to new type).
//
// Example usage:
//     bool ok;
//     type_descriptor_t const *old_type, *new_type;
//     foo %old = nullptr;
//     tie(ok, old_type, new_type) =
//         invalidate_type(type_descriptor(old), new(std::rich) foo);
//     assert(new_type != old_type && ok);
//
// Another case is for “deregistering” a type, by simply providing
// nullptr to the second argument to invalidate_type:
//
//     tie(ok, old_type, new_type) =
//         invalidate_type(type_descriptor(old), nullptr);
//     assert(new_type == nullptr && type_invalidated(old) && !ok);
//
// The above is suggested for module cleanup time when types only meant
// for module scope should be deregistered and all rich pointers for
// these types should be invalidated.
//
// The third argument to invalidate_type is the invalidation handler
// function. This function is called when a rich pointer of an
// invalidated type is dereferenced. It is passed the old type
// descriptor and a reference to the invalidated rich pointer.
```

```
// The invalidation handler should return true if it should allow
// the application to continue execution and false to force a call
// to std::abort. The default handler throws an 'invalidated_ptr'
// exception which contains the invalidated pointer and a pointer
// to the old type descriptor.
template <class T>
tuple<bool, type_descriptor_t const *, type_descriptor_t const *>
invalidate_type(type_descriptor_t const *, T%,
                function<bool(type_descriptor_t const *, T%&)>);
```

There are only ever at most two versions of a type descriptor for any given registered type.

Rationale

The current state of the art of dealing at runtime of types involves manipulating and traversing statically defined type descriptors. These type descriptors contain information about the types they're supposed to represent (members, inheritance hierarchy, etc.) and are encapsulated with the same type they typically describe. This is useful for determining at runtime for example how to display certain pieces of information.

Runtime Introspection

An example of this use-case is the Qt frameworks' QMetaObject type that describes an object that's a member of the Qt type hierarchy. A QMetaObject represents things like type metadata (information associated with types as a key-value pair accessible at runtime) and methods (descriptions of the methods supported, accessed through an integer index).

The way types are annotated to provide this information is through the use of preprocessor macros to generate boilerplate code that registers this information as a static member of the type. Qt has a rich object system and a means for runtime dispatch of methods and traversal/inspection of types at runtime that's overlaid on top of the normal C++ mechanisms already available.

Another example of how richer runtime type data can be used to improve current implementations is how Google Protocol Buffers (ProtoBufs) use type descriptors to define how a protocol buffer is laid out in memory. This enables the protocol buffer messages to keep data in a serialized form in memory and support accessing this data in an efficient manner. The descriptor mechanism allows for parsing incoming data and determine whether the protocol buffer descriptor supports the data that's been received to construct an object of the correct type.

The down-side for both these approaches are:

- The information is represented as redundant information generated either mechanically (in the case of Google ProtoBufs using a proto compiler) or by hand (in the case of custom Qt types).
- Both mechanisms are very rigid and cannot trivially be updated dynamically. The static nature of the information generated does not lend itself to graceful upgrades while programs using these libraries are running.
- Most (if not all) of this rich information is largely only useful when working within the framework of these specific type systems. Mixing and matching ProtoBuf messages and Qt QObjects for instance so that information can be rendered in a GUI or streamed/saved to file typically require glue code to take care of the manual conversion from one type system to another.

An example of how a Google Protocol Buffer (protobuf) serialization mechanism would work is shown below as a function that takes any protobuf object and serializes it to an output stream.

```
// We want to ensure that the pointer we're going to take does
// point to an object statically derived from proto::Message. For
// this we leverage the normal C++ static rules for inheritance
// and polymorphism.
namespace proto {
  bool serialize(Message %m, std::ostream &os) {
    // In here we can then inspect the type associated with the
    // pointer m reflecting the runtime representation.
    type_descriptor_t const *type = type_descriptor(m),
                            *msg_type =
                                type_descriptor<Message>(nullptr);

    // We then traverse the members of the protocol buffer and
    // perform a generic lookup of the data so we can encode
    // properly into the output stream.
    std::string buffer;
    for (field_descriptor_t const *field : type->members) {
      if (field == nullptr) break;
      if (!m->get_member(field->name, &buffer)) return false;
      os.write(s.data(), s.size());
    }
```

```
    return true;
  }
}

// For exposition and completeness, we provide an example implementation
// of proto::Message's get_member function and enclosing scope using
// rich pointers and introspection functionality:
namespace proto {
  class Message {
    char *buffer_;  // dynamically initialized by derived classes
    static std::map<std::string, type_descriptor_t const *> registry;
   protected:
    explicit Message(size_t derived_size)
      : buffer_(new (std::nothrow) char[derived_size])
    {}
   public:
    virtual bool get_member(std::string const &name,
                            std::string *buffer) {
      type_descriptor_t const *type =
        registry[type_descriptor(rich_cast<Message%>(this))->name];
      bool ok = false;
      size_t offset = 0;
      field_descriptor_t const * field = nullptr;
      tie(ok, offset, field) = get_offset(name, type->members);
      if (!ok) return false;
      size_t size = field->type->size;
      buffer->assign(buffer_ + offset, buffer_ + size);
      return ok;
    }
    virtual ~Message() {
      delete [] buffer_;
    }
  }
}
```

Fully Dynamic Extension

For network server applications there have been two major ways of achieving dynamic upgrades of a running system. The way the Apache Web Server does it as an example is to define a very strict protocol for how dynamic shared objects are described and behave -- all extensions are implemented through this system which also relies on the dynamic linker to resolve symbols when new modules are loaded into a running system. The other way is done by many games and dynamic systems is to embed a dynamic language runtime (like Lua) so that the parts that are meant to be upgraded at runtime can be implemented in a fully dynamic programming environment that supports this functionality natively without having to bring the whole application down.

Both these approaches are valid but they have their own trade-offs.

The dynamic module approach relies heavily on external dynamic linking solutions that have traditionally only been implemented in C and support only a C application binary interface (ABI). Although it is possible to hide the C++ implementation behind a C interface it limits the integration possibilities between host applications and dynamic modules. Furthermore this facility is largely platform dependent and is not standardized across vendor implementations.

Embedding a dynamic language runtime has the same limitations as typically the interaction between the host application and the embedded runtime are limited. The types generated or used in the context of the runtime typically have to be converted programmatically into something that the host application explicitly understands. The performance and maintenance costs of systems that do use embedded runtimes are also non-trivial.

Here we show one approach to adding dynamically registered types as plug-ins to a hypothetical server using rich pointers:

```
// We want to have a global registry of handler objects mapped to URLs.
static map<string, pair<void%, function<void<string>>>> handlers;

// Then we define a way for a new module to register completely new
// objects at runtime:
bool register_handler(std::string const &url,
                      void %handler,
                      bool replace) {
  // Here we can determine whether the object being registered as a
  // handler supports the required API (at runtime!)
  type_descriptor_t const *type = type_descriptor(handler);
```

```
  if (type == nullptr) return false;

  bool compliant = true;
  type_descriptor_t const *string_type =
    type_descriptor(rich_cast<std::string%>(nullptr));
  function<void(string)> f;
  for (method_descriptor_t const *method : type->methods) {
    if (method == nullptr) break;
    compliant = compliant || (
      equals(method->type->name, “get”)
      && method->access_type == method_descriptor_t::PUBLIC
      && length(method->type->args) == 2
      && method->type->result_type == nullptr
      && method->type->args[1] == string_type);
   if (compliant && !f.get()) f =
     bind(method->type->callable, f, _1);
  }
  if (!compliant) return false;
  if (!replace && handlers.find(url) != handlers.end()) return false;
  handlers[url] = make_pair(handler, f);
  return true;
}

// When we’re ready to handle a URL, we do the following:
bool handle_url(std::string const &url, std::string const &request) {
  auto it = handlers.find(url);
  if (it != handlers.end()) {
    try {
      (*it)(request);
    } catch (...) {
      return false;
    }
    return true;
  }
  return false;
}
```

Runtime Upgrades and RPC

In the realm of distributed systems programming the remote procedure call (RPC) pattern is very popular and many implementations rely on a mix of static types and dynamic structures to represent types. CORBA services rely on a very rigid client and server interface described using a domain specific language to define the interface by which clients can interact with the server. There are also a number of solutions for web services like SOAP (using XML for encoding RPC calls) and recently more popular REST interfaces (leveraging HTTP semantics and usually passing data encoded in JavaScript Object Notation (JSON)).

CORBA and SOAP services are typically mechanically generated using compilers that generate stubs later filled in by developers. These however suffer from the API versioning problem largely because there's no way for the server to communicate with the clients the existence of new types and methods. Instead the coupling between the client and the server APIs require upgrades to be synchronized or in some cases have costly deprecation paths for client APIs that require the service to keep supporting an old version and the new version at the same time.

For web services written in C++ that use JSON as the data interchange format between client code (typically HTML pages with JavaScript running on web browsers) and server code, the only way to treat the data coming from the client is to treat it as a stream of bytes that are parsed and maybe generate statically typed objects to manipulate in the server. The other problem has to do with streaming hand-rolled or dynamic types as JSON or another data interchange format like XML.

The following example relies on the notion of a JIT compiler hosted by an application that allows for dynamically generating types and objects that can be referred to using rich pointers.

```
// The following function takes some arbitrary input and returns
// a function pointer to a compiled function that returns objects
// via a rich pointer and takes context through a rich pointer
// as well.
function<void%(void%)> Compile(string const &input, void %data);

// Given the example for the network server, let's consider an object
// that exposes its methods as RPC calls.
class HelloWorldRPC {
  string url_;
 public:
  explicit HelloWorldRPC(url const & url)
```

```
  : url_(url) {}

  void get(string const &input);
 private:
  void update(string const &input);
};

void HelloWorldRPC::get(string const &input) {
  HelloWorldRPC %self = rich_cast<HelloWorldRPC%>(this);
  istringstream tokens(input);
  string function;
  tokens >> function;
  type_descriptor_t const *this_type =
    type_descriptor(self);
  for (type_descriptor_t::method_descriptor_t const *method
       : this_type->methods) {
    if (method == nullptr) {
      cerr << “Cannot find function ”
           << function << ‘\n’.
      return;
    }
    if (function == method->type->name) {
      if (length(method->type->args) != 2) {
        cerr << “Cannot invoke function ”
             << method->type->name
             << “ for bad signature.\n”;
      } else {
        // The following call will follow C++ member access rules.
        // In case the calling scope (determined by self) is not
        // the correct type or is not a friend scope, then the call will
        // throw ‘bad_function’.
        method->type->callable(self, input);
        return;
      }
    }
  }
```

```
}

void HelloWorldRPC::update(string const &input) {
  // In here we then compile the input and register a new handler
  // that is yielded by the returned function.
  HelloWorldRPC %self = rich_cast<HelloWorldRPC%>(this);
  function<void%(void%)> factory = Compile(input, self);
  if (factory.get()) {
    // Replace the old handler with the new one!
    auto old = handlers.find(this->url_);
    if (register_handler(this->url_,
                         factory(),
                         old != handlers.end())) {
      if (old != handlers.end()) delete it->first;
    }
  } else {
    cerr << "Cannot update, Compile complains.\n";
  }
}
```

The proposed feature enables for easily maintaining runtime types and allowing for self-updating applications that host fully dynamic runtimes.

Implementation Hints

This section provides information about how we could implement rich pointers in the context of C++11. We approach it by first writing a library solution that approximates the functionality using available language features in C++11. We then show what the limitations of the language and runtime are along the way and work towards removing the limitations.

Library Approach: `rich_ptr`

Here we present a first crack at providing rich pointer support using a template we'll call rich_ptr. The goal of rich_ptr is to provide functionality for the definition and registration of types and objects of these types in the same object. Instances of rich_ptr will follow pointer semantics similar to what shared_ptr provides without the reference counting capability.

The interface for rich_ptr is shown here:

```
// Usage:
//   std::rich_ptr<foo> p { new foo };
template <class T>
struct rich_ptr {
  rich_ptr() = default;
  explicit rich_ptr(T *ptr);
  rich_ptr(rich_ptr const &other) = default;
  rich_ptr(rich_ptr &&other) = default;
  rich_ptr& operator=(rich_ptr) = default;

  // Handle the case for when a normal bare pointer is assigned
  // to a rich pointer.
  template <class U> rich_ptr& operator=(U *);

  ~rich_ptr() = default;
  T& operator*() const;
  T& operator->() const;

  // We also want rich_ptr<T> to be convertible to T*.
  operator T*() const;
 private:
  T *ptr_;
  type_descriptor_t const *descriptor_;
  static type_descriptor_t const *unique_descriptor;
  friend template <class V> type_descriptor_t const *
    type_descriptor(rich_ptr<V> const &);
  friend template <class V> bool
    type_invalidated(rich_ptr<V> const &);
  friend template <class V> tuple<bool, type_descriptor_t const *>
    register_type();
  friend template <class V> void
    swap(rich_ptr<V> &, rich_ptr<V> &);
};
```

The minimal interface is by design meant to mimic how pointers work. There are no special member functions except for overloads to the dereference operator and the member access

operator.

We then supply a means to programmatically defining type descriptors following the already presented type descriptors by hand. An example is shown below:

```
template <class T>
tuple<bool, type_descriptor_t const *>
register_type() {
  static_assert(false, “No explicit definition for type provided.”);
  return make_tuple(false, nullptr);
}

// Let’s annotate the type and wrap it in macro’s so that we can
// programmatically generate the explicit overload for the register_type
// template function.
DEFINE_REGISTERED_CLASS(foo)
  REGISTERED_CONSTRUCTOR(foo()) = default;
  REGISTERED_DESTRUCTOR(~foo);
  REGISTERED_MEMBER(void bar());
  foo(foo &&) = delete;
  foo(foo const &) = delete;
  foo& operator=(foo) = delete;
  PRIVATE_MEMBER(int a_);
  PRIVATE_MEMBER(string b_);
END_REGISTERED_CLASS_DEFINITION(foo);

// We provide explicit overloads and definitions for a given type.
// This is generated by the macros above right where the class is
// defined (in END_REGISTERED_CLASS_DEFINITION).
template <>
tuple<bool, type_descriptor_t const *>
register_type<foo>() {
  static type_descriptor_t const foo_type {
    /* Defined as shown earlier in the paper. */
  };
  // We then duplicate this definition in the rich_ptr<T> static
```

```
  // unique_descriptor.
  rich_ptr<T>::unique_descriptor = &foo_type;
  return make_tuple(true, &foo_type);
}

// We then call the explicit registrations for types we want to use in
// main(), or use static initialization of globals to rely on invoking
// the register_type<...>() function for all registered types.
int main(int argc, char *argv[]) {
  register_types<foo>();  // A variadic template function that takes
                          // a list of types to actually register at
                          // the start of main.
  // as main would normally be.
  return 0;
}
```

Library Approach Limitations

There are a few problems in the above approach:

- The syntax of the macros is very explicit and too verbose and deviates too much from the normal C++ syntax for class definitions.
- Because you have to consciously annotate the type you're registering, this makes it hard to manage especially if the type's definition is not something you can change (i.e. when using a third-party provided library).
- This will not properly handle dynamically loaded libraries at runtime and will be at the mercy of the implementation and platform.

With a library-only implementation we cannot do the following reliably:

- Safe runtime type invalidation. This will require two things:
    a. A standardized registry of types available and managed at runtime.
    b. Well defined module initialization/cleanup semantics and interfaces.
- Seamless invalidation at runtime. Because of the linkage and concurrency issues introduced by standard-allowed optimizations, it is not guaranteed that library-defined invalidation routines can be protected. The worst case scenario is that code paths and branches that can be optimized appropriately using normal pointers as opposed to rich pointers are in jeopardy of behaving much poorly.
- Safe runtime type creation. For JIT compilers at the moment, only code and data that's

accessible through opaque pointers (void *) and rigidly-defined function pointers that have primitive return types can be integrated into the hosting runtime context. New types cannot be safely registered and invalidated without explicit runtime support of type registration.

- Automatic descriptor generation. With the library approach types need to be explicitly registered and annotated. The compiler on the other hand has all the information it needs and can generate the type information at compilation time and can do so only for types that are used in rich pointers and types pointed to by rich pointers.

The above motivations are the reason why we're proposing that rich pointers be added to the language so that we can bring more of the static information about the program at runtime through standardized interfaces across platforms.

Language Feature: "`type %`"

Changes to the actual standard on the wording of the proposed language feature is yet to follow. At the moment this section provides the specific guarantees and requirements for the rich pointer type and what the usage semantics are at a high level.

Guarantees and Requirements

A rich pointer to a type T shall support complete value semantics and shall be treated as a primitive data type. T can be any primitive type including void as well as any valid user-defined type.

A rich pointer will not support normal pointer arithmetic.

A rich pointer's size is guaranteed to be at most twice the size of a normal pointer.

Dereferencing rules for normal pointers apply to rich pointers fully.

When a rich pointer's value is assigned to a normal pointer only the memory location and potentially current runtime type information is preserved (if supported by the implementation).

When a normal pointer's value is assigned to a rich pointer a runtime check for type suitability based on available RTTI information (if supported by the implementation) is used to determine the type descriptor to be associated with the rich pointer. This check is performed implicitly using `rich_cast<...>`.

Special Rich Pointer: `void %`

A void % is meant to still carry whatever type descriptor is associated with whatever rich pointer

is assigned to it. The following constructs are provided as examples for the special rich pointer.

```
// A void % can point to any rich pointer value.
foo %p = new (std::rich) foo;
void %super = p;  // No cast required, type descriptor is carried.
assert(type_descriptor(p) == type_descriptor(super));

foo %r = rich_cast<foo%>(super);
if (r == nullptr) {  // means it failed.
}

// For virtual inheritance rules, we also rely on existing RTTI.
base *b = new derived;  // assume 'class derived : public base'
void %super = rich_cast<base%>(b);
assert(type_descriptor(rich_cast<derived%>(b)) ==
       type_descriptor(super));  // preserve the type descriptor.

// We may potentially lose the type in case the value comes from
// void *, in which case we can't reliably guess at runtime what
// the type of the object at a given address is.
derived *d = new derived;
void *v = d;  // Okay, normal pointers.
void %super = v;  // Still okay, but...
assert(type_descriptor(super) == nullptr);  // We don't know the type.
```

Casting: `rich_cast<...>`

When casting from one type of rich pointer to another a call to `rich_cast<destination_type%>` is required. This does two things:

- Force the compiler to generate the type descriptor for `destination_type`.
- Checks whether the conversion is valid following the same rules that `dynamic_cast<destination_type*>` follows.

An implicit rich_cast<...> is called on a pointer when a normal non-void pointer is assigned to a void %. More concretely:

```
// These two statements are equivalent.
```

```
void %f = new foo;  // notice no std::rich in call to new.
void %g = rich_cast<foo %>(new foo);
assert(type_descriptor(f) == type_descriptor(g));
```

Dynamic Construction: `new (std::rich)`

The special form of the new operator intends to differentiate the normal static construction of objects of a given type using the static definition of type from the most-up-to-date runtime definition of the same type. This is so that the semantics of existing code using normal calls to new are not affected by the new semantics of this special form of new.

With this different construct, it is expected that compilers issue diagnostics in the following potentially dangerous situations:

```
// Assigning the result of new (std::rich) to a normal pointer causes
// the type descriptor to be dropped immediately. The newly constructed
// object's layout may then be different from the statically defined
// version of the type (i.e. crossed module boundaries or the static
// definition of the type has been already invalidated).
foo *f = new (std::rich) foo;

// Using placement new (std::rich) on a statically-sized buffer. This
// may cause buffer overflows when the definition of a type changes at
// runtime.
char b[sizeof(foo)];
new (b, std::rich) foo;
```

Runtime Integration

There are a lot of areas for which determining the integration between runtimes and type information. The basic requirements for rich construction and type information management at runtime are:

- Means for registering, updating, and invalidating types at runtime.
- Standard interface for loading dynamic modules.
- Algorithms for easily inspecting types at runtime.

Each of these areas identified are beyond the scope of this paper, but are implied in the semantics for usage by the proposed feature.

References

N2316 - C++ Modules (revision 5) by Daveed Vandevoorde

In the C++ Modules proposal, the initialization order and means for introducing a way for formally supporting dynamic libraries (as introduced by *N1400 - Toward Standardization of Dynamic Libraries by Matt Austern*).

N1976 - Dynamic Shared Objects Survey and Issues

This paper largely goes through the various papers regarding the topic of shared objects and dynamic libraries.

N1751 - Aspects of Reflection in C++ by Detleff Vollmann
N1775 - A Case for Reflection by Walter Brown et al

These two papers describe generally how Reflection and Introspection would be approached, but lacked specific implementation details that this paper partly provides.

Acknowledgements

Thanks to the following people who have contributed their feedback on the paper and the overall concept: David Abrahams, Roman Perepelitsa, Jason Bolla, Lawrence Crowl, and Matt Austern.