Type Name Strings For C++

Document Number: ISO/IEC JTC1 SC22 WG21 N3437=12-0127

Author: Axel Naumann `axel@cern.ch`

Date: 2012-09-24

Contents

Abstract

Several proposals for introspection-oriented extensions are on the table. All of them miss a fundamental ingredient: representation of type names as strings.

This paper describes use cases of type introspection and reflection based on CERN's experience, where the existing proposals fall short, and which features an implementation that could facilitate the use cases discussed. It argues that such an extension could turn C++ into a much more versatile language, and help

with notoriously difficult areas such as serialization, language binding, runtime documentation and signal-slot.

This paper is not a proposal; it is meant to show which parts of introspection might benefit from additional features as compared to current C++. Its purpose is to discuss a fundamental step, not a Grand Theory of Everything. Depending on the relevance of this topic outside our field, we'd be happy to prepare a proposal with a prototype implementation.

1 Motivation

The connection between types and strings is strong and simple: type names play a key role in C++ as they are used for type identification. Given that a (fully qualified) type name is traditionally unique in a program, they can be used to identify a type also at runtime. Traditional occurrences of this use are plug-in mechanisms ("You want type **A**? Let me load libA for you"), signal-slot ("Function **func** in class **Slot** should now be called!"), serialization ("the next bytes to read belong to an object of type **Klass**"), language binding ("the python function argument points to an object of type **CxxKlass**"), and even dynamic documentation ("let me show you the documentation on the data type **Fancy**"); they are discussed below.

With the current standard, implementations are platform-dependent due to **type_info**::**name**() providing platform-dependent results. If on the other hand they are based on template meta-programming, they are difficult to access outside the type system (for instance in a persistent type description database). Combined, they cause a lot of overhead to the above use cases. Instead, some implementations generate code to annotate types, where the annotation gets attached to the types through preprocessor macros defining static data members. None of that is especially beautiful, nor simple to implement, nor easy to use.

This note will analyze several key use cases in light of their need for a string representation of type names. Given these use cases, a (short) list of required features is compiled.

2 Use Cases

2.1 Serialization

Serialization requires the description of the in-memory layout of a data type. C++ does not provide a tool for this; while this would be good to have, it is beyond the scope of this note.

All useful (de-)serialization routines are written based on extensible type information databases. Examples include:

- Google's Protocol Buffers `http://code.google.com/p/protobuf`,
- Boost::serialization `http://www.boost.org/libs/serialization`,

- Qt QDataStream `http://doc.qt.digia.com/4.7/qdatastream.html` or
- CERN ROOT [1] `http://root.cern.ch`

They allow additional types to be defined and implementations to be provided, e.g. through shared libraries, without a change to the underlying deserialization routines. What all tools depend on, though, is the identification of their element in their type description database given an object. If the serialized representation of an object of type **std**::**vector**<**long int**> comes in, e.g. through a network socket, the deserialization routines need to know which type description this corresponds to. Using the string representation of the type name for identifying the type introspection data is thus a natural choice.

The approaches of Boost and Qt serialization bind the serialization to the type; here, the serialization library depends on the types it can serialize. Protocol Buffers even defines the types itself, by generating their C++ definitions. Additionally, all (but an extension of Protocol Buffers) lack a key feature of ROOT serialization, which is self description: ROOT files contain a definition of the data layout (which is defined at runtime), which in turn requires type names as the only way of identifying types across implementations, binaries, operating systems etc. This enables reading the file without the types' definitions loaded into the binary (a mock-up object can be constructed from the data layout), and schema evolution, where the type definition on disk and in the binary are different.

2.2 Type Introspection

For the purpose of this paper, type introspection is defined as the ability of an object to identify its type towards a client that is not aware of the type, i.e. the client's code does not contain a definition of the object's type.

Type introspection enables remote procedure calls: if the type string for an object can be retrieved in one implementation and be looked up in another implementation, then applications can agree on a type's member functions in a dynamical way, by passing the ingredients of their signatures. A server might offer "clients can call a function named **Func1** passing a serialized representation of an object of type **A**; I will return the serialized representation of type **B**.". C++ clients will be able to determine whether that function can be called given a parameter value list.

A less obvious use case for type introspection is data exploration. It's a wonderful tool for complex data structures: users (or developers) can explore objects and their relationship, through a mixture of runtime type evaluation, type introspection data and documentation identified by the type name at runtime.

It helps to identify which bits of data users are interested in and how these data behave. Suppose the data to analyze by a program contains manufacturing quality data. The algorithm is supplied at runtime. To do that, the user can see which data fields exist: for instance the manufacturing site might have a data member that specifies the country of the location – maybe a good candidate for

a relevant parameter in quality studies. This might sound surreal, but it is very close to how physicists do data analysis at CERN. Data exploration reduces the gap between code development, code execution and documentation, which can turn development data-centric instead of type-centric.

2.3 Language Binding

An introspection database enables type descriptions to cross language borders: a structure containing two members of type **float** and one containing a collection of **int** can be defined in most languages. Conversions can be (and are) done dynamically at runtime, based on type introspection data. This is especially useful when binding to a language with a dynamic type system. Also here, the key prerequisite is the identification of the type, to be able to retrieve the type description for a given object from a persistent type introspection database.

3 Features

As mentioned before, these use cases require only a minimal (and minimally invasive) set of features.

3.1 Type Name Representation

It might be possible to standardize a canonical type name for each type. For users this would be the optimal case: removing ambiguities and the need to convert or parse type name strings.

Alternatively, the implementation must be able to retrieve the **type_info** object for (a defined subset of) a type's fully qualified type name aliases. For the purpose of clarity, this paper calls this function

```
static const std::type_info*
   std::type_info::from_name(const char*)
```

This would be accompanied by a new name retrieval function, which for clarity is called

```
const char* std::type_info::id_name()
```

It would return a valid C++ type name, i.e. a name that is not mangled, and that can serve as input to **type_info**::**from_name**() for all other implementations.

type_info::**from_name**() requires the type to be defined, else its **type_info** object is unavailable. That is likely an unnecessary restriction, especially for runtime features that can provide a definition given the type name, for instance plug-in systems or just-in-time compilers. It would thus be beneficial to have a conversion of a type name into the type name returned by **type_info**::**id_name**(), as if its **type_info** object existed. Again for improved clarity, this paper refers to it as

```
static const char*
   std::type_info::to_id_name(const char*)
```

Given **type_info**::**to_id_name**(), a simple **type_info** retrieval function expecting the string returned by **to_id_name**() as parameter would suffice. For the purpose of clarity, this paper calls this function

```
static const std::type_info*
   std::type_info::from_id_name(const char*)
```

from_name() and **to_id_name**() do not need to be able to accept all possible versions of type names as input, but only a defined, standardized subset. The subset definition should enable the implementation to treat identifiers as a set of tokens, without knowledge about their nature. A possible subset could be all type names with all default template arguments removed, with all typedefs resolved, with signedness and unsignedness specified where applicable, and all integer types that optionally allow it to end on **int**, and all expressions (for template arguments) evaluated. Thus, "std::vector<long*, std::allocator<long*>>" would not be part of the allowed input subset; but "std::vector< signed long int * >" would be part of the allowed input subset. Ideally all that is left for an implementation to do is to normalize the location of the cv-qualifiers and spacing.

Some type names will be implementation-defined. Clients are thus forced to deal with "synchronizing" the type names across different implementations. This is not a deterioration of today's situation for clients.

As examples for above definition of allowed input, **n1**, **n2** and **n3** must compare equal:

```
const std::type_info &n1 = *std::type_info::from_name(
   "std::vector<const volatile long int*>");
const std::type_info &n2 = *std::type_info::from_name(
   "std :: vector < const volatile long int * >");
const std::type_info &n3 = *std::type_info::from_name(
   "std::vector<volatile const long int * >");
```

Combining the different type name retrieval paths, the two strings **id1** and **id2** must string-compare equal for all type names **tn** in the subset of allowed type names:

```
const char* id1 =
   std::type_info::to_id_name(tn);
const char* id2 =
   std::type_info::from_name(tn)->id_name();
const char* id3 =
   std::type_info::from_id_name(id1)->id_name();
```

To make the new features available, one of these alternatives could be implemented:

- A canonical type name defined for all types by the standard; or
- **type_info**::**to_id_name**(), **type_info**::**from_id_name**() and **type_info**::**id_name**(), where the "id name" is standardized except for optional white space and commutative keywords, where the existence (white space) and order (keywords) is fixed by the implementation.

type_info::**from_name**() is thus optional and not a prerequisite for the features described here.

3.2 Runtime Type Information

The name for types must be accessible e.g. through the **std**::**type_info** object for a type, as outlined above for **std**::**type_info**::**id_name**(). The existing functions for retrieving the **std**::**type_info** given an object are sufficient for this discussion.

3.3 Impact On Use Cases, Relationship With Type Introspection

Currently, given a **type_info** object, one cannot retrieve a key that can be used in a universal, implementation-independent way to identify the type description in an external type description database, which is the fundamental ingredient for all use cases described above. The usual reaction is instead to intrusively annotate all supported types by a key, e.g. through a static data member.

Alternatively, to use the **type_info** object as key, compiled code must define a connection between the **type_info** object and a static key. That compiled code is loaded into the binary at runtime. Given a **type_info** object, the static key is looked up, which in turn serves as the key in the type description database.

Both could be dramatically simplified. Standardization of a canonical type name would allow the use of a type name as a key, after mapping implementation-defined type names, as is already done now. Alternatively, all type names used as key in a type description database would need to be replaced by their implementation-specific version by calling **std**::**type_info**::**to_id_name**(**const char***).

Type name normalization is a key ingredient of type description systems. It does not provide answers to "which data members does class A have". But it facilitates the use of existing type description libraries considerably. The features discussed here are independent of whether type introspection and reflection features will be available in a future C++ version, and what form they might have. This note does however point out use cases that are difficult to satisfy with a type introspection implementation that is similar to type traits, i.e. specifically without a string representation.

4 Characterization of Implementation, Cost

A basic implementation of type name parsing would have to be available in the C++ runtime libraries if the type names are not standardized. It would likely incorporate features that already exist in today's compilers. There is no CPU time penalty for clients not using this feature, the only cost is an increased runtime library size.

Additionally, a program would need a type name database at runtime, if runtime type information is requested, which is the only way to extract a type name from a type. This could be done by extending the current data structures used to represent **std**::**type_info**.

The amount of string constants introduced by this interface is not negligible. An example, one of ROOT's libraries has about 24k non-blank lines in header

files defining more than 600 types; their type names (in an arbitrary notation) consume 12k bytes in total (including the string-terminating \0). The storage cost could be reduced by storing the types' unqualified name and the types' declaration scope, and constructing the fully qualified name on demand.

5 Comments on Existing Proposals

Existing proposals and implementations in the Standard Library offer characterization of types at compile-time [2], [3]. This is not sufficient for the use cases outlined in the paper: they provide a wonderful tool to describe types within one translation unit, and to build reflection data in a static way. But they do not provide a way to make this description persistent with a unique key: the type name.

Other proposals and comments go far beyond what is discussed here, describing possible implementations of reflection databases. They either require language changes [4] or show the complexity of such an endeavor [5]. Both (though not explicitly) mention type name strings, another proof of the relevance of this topic for reflection. The scope of this note is much reduced, though, making it hopefully more realistic to agree on an implementation, and providing a more focused and basic seed for future extensions in the region of reflection for C++.

6 Summary

While C++ is providing more and more compile-time-centric features, string-based type identification enables C++ to cover more use cases. Zero-terminated strings are a common denominator for many languages, or at least for their binding libraries. Using them for type communication across languages will thus also help language bindings in a simple, straight-forward way. This paper has shown that type name strings can dramatically simplify problems that are currently impossible to solve without crude external scaffolding.

Acknowledgments

Thanks to the following people for their feedback and discussions: Philippe Canal (Fermilab), Vassil Vassilev and Jakob Blomer (both CERN).

References

[1] Ilka Antcheva et al, *ROOT - A C++ framework for petabyte data storage, statistical analysis and visualization.* Computer Physics Communications; Anniversary Issue; Volume 180, Issue 12, December 2009, Pages 2499-2512.

[2] David Vandevoorde, *Reflective Metaprogramming in C++* N1471=03-0054, 2003

[3] Detlef Vollmann, *Aspects of Reflection in C++*, N1751=05-0011, 2005

[4] Dean Michael Berris, Matt Austern, Lawrence Crowl, *Rich Pointers*, N3340=12-0030, 2012

[5] Walter E. Brown, Philippe Canal, Mark Fischler, Jim Kowalkowski, Pere Mato, Marc Paterno, Stefan Roiser, Lassi Tuura, *A Case for Reflection*, N1775=05-0035, 2005