Xcpp Reflection

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

The CEDA front end compiler Xcpp supports reflection of classes, interfaces, functions and other constructs defined in the source code. This paper presents some motivating examples and describes the syntax.

1 Overview

In this article *reflection* refers to recorded information about *types* such as classes and interfaces defined in the source code processed by the CEDA Xcpp front end.

2 Purpose

There are many reasons why reflection can be very useful. There are numerous frameworks that can be closed for change because they are able to deal with arbitrary software components described with reflection information. An application programmer can write a simple software component and the system provides a range of capabilities such as persistence or cross language support for free. Here are just some of the facilities that can utilise reflection information:

- Reflected classes may support dynamic creation of instances of that class, supporting the *Abstract Factory* design pattern [1]. A client can be passed a reference to a reflected class and create instances of that type without any compile time dependency on that type.
- *Visitor* Design pattern. In the Java language, reflection has been used to get rid of the accept methods [2], or to get rid of extra visit methods.
- Dynamic Dispatch : A client invokes a function with a signature that is only determined at run time using reflection information (not to be confused with dynamic calls based on virtual methods where the function signature is known at compile time).
- Middleware support : Reflection information can be used to automatically generate client side proxies and server side stubs for reflected functions (or perhaps an entire reflected interface), allowing for function arguments and return values to be automatically marshaled over the wire.
- Cross language support (e.g. to the *Python* language or to .NET)
- Logging of reflected datatype values.
- GUI generation:
  - Reflected data types can be automatically viewed or edited. E.g. .NET uses this for a property grid.
  - Reflected functions can be mapped to buttons or menus in GUI systems, even using dialogs to edit the inputs (i.e. in-parameters) if any, and to present the results (i.e. the out-parameters) if any.
- Binary or XML serialisation of reflected datatypes.
- Mapping between alternative data representations.
- Persistence. Reflected datatypes can be made to automatically persist on secondary storage.
- Replication. Reflected datatypes can automatically be replicated in a distributed system, and support automatic synchronisation by sending changes on the data as serialised deltas between sites.
- Collaborative editing. Reflected datatypes that are replicated in a distributed system can support real-time interactive collaboration.
- Versioning, merging and branching of reflected datatypes.
- Query forms can be automatically generated to allow users to issue database queries.
- A query engine can execute a query making use of the reflection on the datatypes.

3 Syntax

As a general rule, all the Xcpp language extensions involving '$' have reflection information generated automatically. For example, each of the following constructs are reflected:

```
// Functor
$functor int32 MyFunctor(int32 x);

// Typedef
$typedef float64 Mass;

// Class/struct
$class X
{
public:
  $int32 GetX() const { return x; }
private:
  $int32 x;
};

// Interface
$interface Ix
{
    void foo();
};
```

```
// Global variable
$var int32 x;

// Global function
$function void bar();

// Tuple
$tuple Point
{
  int32 x;
  int32 y;
};
```

Note therefore that reflection is *optional*. For example classes and functions written in normal C++ are not reflected.

4 Reflection Registries

cxObject.dll has global, transient, threadsafe registries for each of the following:

- Classes (which includes structs and tuples)
- Interfaces
- Functors
- Enumerated types
- Variants [not implemented yet]
- Typedefs
- Global functions
- Global variables

Each registry supports a request for the reflection information of an item by specifying its fully qualified name (as an ASCII string). E.g. one can request the reflection information for the interface named `ceda::IObject`.

Each of these registries supports iteration over the items. Note that it is possible to think of the items as forming a tree structure according to the C++ namespaces.

Each application DLL registers its items when it is first loaded, and they are unregistered when the DLL is unloaded. Therefore at any point in time these global registries provide information about what is currently available in the executing process.

For advanced users that want more dynamic control over the registries, there are functions to directly register and unregister items.

5 Namespace registry

All the reflected interfaces, classes etc are also registered in a single *namespace tree*. Each namespace node registers all items under that node in a single map keyed by a string identifier. This means that it is necessary for all items to have unique names - even different types. This restriction is not conventional in C++ (which for example, within the same namespace allows for a class to have the same name as a global variable). This restriction is made in the interests of easing cross language support, such as with the Python language.

6 Metadata in reflection information

Some authors refer to all reflection information as *metadata*. However in this article we restrict that term to the user defined, auxiliary part of the reflection information not representing C++ types.

Xcpp allows arbitrary *metadata* to be recorded against reflected types. For example:

```
$typedef uint32 Flags : [hex];
```

The metadata is recorded as a comma separated list of *terms* enclosed in square brackets. An example of a term is an identifier like `hex`. The meaning and purpose of the metadata is application defined. In this case the metadata `hex` has been applied to the type `Flags`, presumably to control the appearance when values are printed by frameworks that use the reflection information. As far as the C++ type system is concerned, `Flags` is just an alias for a `uint32`. However the reflection system treats it as an independent type with its own recorded reflection information.

The following example defines a type called `Mass`, that has kg units and the minimum is 0 meaning that negative masses are not permitted.

```
$typedef float64 Mass : [unit("kg"),min(0)]
```

A GUI framework can use this metadata to automatically display the units and to validate user entry (i.e. ensuring that the entered value isn't negative). One of the aims in CEDA is to promote GUI frameworks that can be used to automatically present and edit arbitrary data models. Meta-data on data model fields allows for fine grained control over many details, including layout management, choice of edit control (e.g. slider versus text box), data validation etc.

In general metadata is applied with a colon followed by a list of metadata items in square brackets. The grammar is simple but extremely flexible. E.g.

```
$typedef float64 Pressure :
  [
    unit("kPa"),
    range(50,300),
    levels
    (
      [
        [name("low"), max(75),
          action(email(supervisor))],
        [name("medium"), range(75,100) ],
        [name("high"), range(100,250)],
        [name("fatal"), min(250),
          action( email(system_controller) )]
      ]
    )
  ];
```

The meaning and purpose of metadata is not defined by the core ceda framework. It is simply a facility to be used (or abused) by application programmers as they see fit.

A functor consists of an identifier, optionally followed by a comma separated list of items enclosed in round brackets. Note that `hex` is a functor with arity zero and is equivalent to `hex()`.

The grammar for the C++ type system is recursive. For example, given any type one can typically form arrays over that type. We therefore say that types can *nest*. Metadata can be applied at all levels within a nested type. As an example, there are two places where we can apply metadata in the following

```
$typedef int32* Pointer;
```

In the following `base(9)` applies to the `int32` and `hex` applies to the overall `Pointer`.

```
$typedef int32:[base(9)] *:[hex] Pointer;
```

Metadata can be applied to a function's return type, each of the argument types and also to the function as a whole. For example

```
$function float64:[unit("m/s")]
  GetVelocity(
          float64:[unit("m")] s,
          float64:[unit("m/s^2")] a)
      : [description(
       "Calculate velocity when a stationary "
       "object accelerates at a for "
       "displacement s")]
{
  return sqrt(2*a*s);
}
```

In practice metadata can easily obscure the code, and it is probably a good idea to tend to use typedefs where possible. This also avoids the need to repeat metadata information.

Metadata can be applied to an interface and also on each method in the interface. For example

```
$interface ICDPlayer : [releaseDate(2010,7,23)]
{
  void Play(int32:[range(0,100)] speed);
};
```

Similarly metadata can be applied to a class/struct, and also to each reflected method or member variable.

7 Metadata grammar

The following EBNF grammar (written according to [3]) defines the format supported for meta-data.

```
boolean literal = 'false' | 'true';

literal =
  string literal |
  integer literal |
  floatpt literal |
  boolean literal;

element = list | functor | literal;

list = '[', [element, {',', element}], ']';

functor:
  identifier,
  ['(', [element, { ',', element }], ')'];

metadata = list;
```

where we assume the following symbols are output by a conventional C++ lexical scanner:

Symbol	Example	Description
identifier	`_ax3_111`	Begins with letter or underscore, then any number of letters, underscores or digits.
string literal	`"hello, world\n"`	Double quoted string literal with support for escaped characters as for string literal in C/C++.
floatpt literal	`-10.453e-8`	Floating point number, as per C/C++
integer literal	`-78`	Base 10 integer literal

8 Reflecting the function to write an object to an ostream

In the following class, the `<<print>>` directive tells Xcpp to reflect the `operator<<()` function that writes the class/struct to an `ostream`. This will affect the display of the object in any framework that cares to use the reflection system. For example, it is used by the CEDA Python bindings library.

```
$struct Point <<print>>
{
  $int32 x;
  $int32 y;
};

std::ostream& operator<<(std::ostream& os,
  Point p)
{
  os << '(' << p.x << ',' << p.y << ')';
  return os;
}
```

9 How reflection information is recorded

9.1 Example

The typedef for `Pressure` appearing in section 6 causes the following reflection information to be generated, to be compiled by a standard C++ compiler:

```
// Registration of Pressure
namespace
{
  static const BYTE Pressure_type[] =
  {
    0xe3,0xc1,0x00,0x00,0xa4,0x01,0x00,0xc2,
    0x02,0x00,0xa2,0x32,0x00,0x00,0x00,0xa2,
    0x2c,0x01,0x00,0x00,0xc1,0x03,0x00,0xe4,
    0xe3,0xc1,0x04,0x00,0xa4,0x05,0x00,0xc1,
    0x06,0x00,0xa2,0x4b,0x00,0x00,0x00,0xc1,
    0x07,0x00,0xc1,0x08,0x00,0xc0,0x09,0x00,
    0xe2,0xc1,0x04,0x00,0xa4,0x0a,0x00,0xc2,
    0x02,0x00,0xa2,0x4b,0x00,0x00,0x00,0xa2,
    0x64,0x00,0x00,0x00,0xe2,0xc1,0x04,0x00,
    0xa4,0x0b,0x00,0xc2,0x02,0x00,0xa2,0x64,
    0x00,0x00,0x00,0xa2,0xfa,0x00,0x00,0x00,
    0xe3,0xc1,0x04,0x00,0xa4,0x0c,0x00,0xc1,
    0x0d,0x00,0xa2,0xfa,0x00,0x00,0x00,0xc1,
    0x07,0x00,0xc1,0x08,0x00,0xc0,0x0e,0x00,
    0x0d
  };
  static const char* Pressure_stringTable[] =
  {
      "unit",
      "kPa",
      "range",
      "levels",
      "name",
      "low",
      "max",
      "action",
      "email",
      "supervisor",
      "medium",
      "high",
      "fatal",
      "min",
      "system_controller",
  };
  const ceda::ReflectedTypedef Pressure_typedef =
  {
      "Pressure",
      Pressure_type,
      Pressure_stringTable
  };
  struct SelfRegisterPressure
  {
```

```
    SelfRegisterPressure()
    {
      cxVerify(ceda::gfnRegisterReflectedTypedef(
        &Pressure_typedef) == ceda::NSE_OK);
    }
  } srPressure;
}
```

All the strings have been placed into a *string table*, and the type definition contains all the meta-data using an array of bytes. The latter is referred to as a *byte code*. The byte code uses a very small memory footprint yet can be processed (i.e. decoded) quickly and efficiently.

9.2 Byte code

Reflected types (including auxiliary meta-data) are represented using a byte code. This article presents the grammar using the ISO standard [3] except that instead of terminals representing text enclosed in quotes we instead use terminals representing byte values written in C++ hex notation (e.g. 0x0B for the byte code 11 in decimal). It is also convenient to introduce the following symbols:

Symbol	Description
`int8`	8 bit signed integer represented by a single byte.
`int16`	16 bit signed integer formed by 2 consecutive bytes in the byte code assuming little endian.
`int32`	32 bit signed integer formed by 4 consecutive bytes in the byte code assuming little endian.
`float64`	64 bit double precision floating point formed by 8 consecutive bytes in the byte code assuming IEEE 754.

9.2.1 Basic types

A single byte code value is used to represent the basic types as follows:

```
FT_VOID = 0x00;
FT_BOOL = 0x01;
FT_INT8 = 0x02;
FT_INT16 = 0x03;
FT_INT32 = 0x04;
FT_INT64 = 0x05;
FT_INT128 = 0x06;
FT_UINT8 = 0x07;
FT_UINT16 = 0x08;
FT_UINT32 = 0x09;
FT_UINT64 = 0x0A;
FT_UINT128 = 0x0B;
FT_FLOAT32 = 0x0C;
FT_FLOAT64 = 0x0D;
FT_CHAR8 = 0x0E;
FT_CHAR16 = 0x0F;
FT_STRING8 = 0x10;
FT_STRING16 = 0x11;

basic_type =
  FT_VOID | FT_BOOL |
  FT_INT8 | FT_INT16 | FT_INT32 |
  FT_INT64 | FT_INT128 |
  FT_UINT8 | FT_UINT16 | FT_UINT32 |
  FT_UINT64 | FT_UINT128 |
  FT_FLOAT32 | FT_FLOAT64 |
  FT_CHAR8 | FT_CHAR16 |
  FT_STRING8 | FT_STRING16;
```

9.2.2 Referenced Types

The byte code may reference other (reflected) named types using the fully qualified name, such as `ceda::IObject`. This allows a client that is processing the byte code to use the fully qualified name to look up the relevant reflection registry (see section 4).

Rather than record a string directly in the byte code, the strings are instead assumed to be recorded in a separate *string table*. Furthermore it is assumed that the relevant string table contains fewer that 65536 entries so therefore a 16 bit integer can be used to index into the string table. It is the 16 bit string table index that is recorded in the byte code.

A reference to a named type always begins with a byte value that indicates the "kind" of type that has been referenced (i.e. an indicator for whether it is a class, interface, or enum etc). In more detail, this means the kind of type as it has been declared and therefore recorded in one of the global reflection registries, and therefore may in fact be a typedef (and in that case it is not possible to tell what the type actually is until looking up the reflected typedef registry with the given fully qualified name). Client code that processes byte code will typically treat typedefs as an indirection, and will use the typedef registry to step into the byte code that the typedef in turn designates.

```
FT_CLASS = 0x12;
FT_INTERFACE = 0x13;
FT_TYPEDEF = 0x14;
FT_ENUM = 0x15;
FT_FUNCTOR = 0x16;
FT_OPAQUE = 0x17;
FT_UNION = 0x18;
FT_VARIANT = 0x19;

ref_key =
  FT_CLASS | FT_INTERFACE |
  FT_TYPEDEF | FT_ENUM |
  FT_FUNCTOR | FT_OPAQUE |
  FT_UNION | FT_VARIANT;

string_table_index = int16;

referenced_type =
  ref_key, string_table_index;
```

9.2.3 Unary qualified Types

There are a number of byte code values that prefix a given type. For example, prefixing any given byte code with the value `FT_POINTER = 0x1C` changes the type to instead be a pointer to that type. E.g.

C/C++ type	Byte Code
`int32`	`[0x04]`
`int32 *`	`[0x1C 0x04]`
`const float64 *`	`[0x1C 0x1B 0x0D]`

```
FT_VOLATILE = 0x1A;
FT_CONST = 0x1B;
FT_POINTER = 0x1C;
FT_REFERENCE = 0x1D;
FT_INTERFACE_POINTER = 0x1E;
FT_PREF = 0x1F;
FT_CREF = 0x20;
FT_VECTOR = 0x21;
FT_DEQUE = 0x22;
FT_LIST = 0x23;
FT_SET = 0x24;
FT_BAG = 0x25;
FT_DYNARRAY = 0x26;

unary_type_qualifier =
  FT_VOLATILE |           (* volatile T *)
  FT_CONST |              (* const T *)
```

```
  FT_POINTER |            (* T* *)
  FT_REFERENCE |          (* T& *)
  FT_INTERFACE_POINTER |  (* ptr<T> *)
  FT_PREF |               (* pref<T> *)
  FT_CREF |               (* cref<T> *)
  FT_VECTOR |             (* xvector<T> *)
  FT_DEQUE |              (* xdeque<T> *)
  FT_LIST |               (* xlist<T> *)
  FT_SET |                (* xset<T> *)
  FT_BAG |                (* xbag<T> *)
  FT_DYNARRAY;            (* T[] *)
```

9.2.4 Meta-data

Any type can be qualified with auxiliary, user defined meta-data, by prefixing the byte code with the metadata information. E.g.

C/C++ type	Byte Code
int32	[0x04]
int32 : []	[0xE0 0x04]
int32 : [false]	[0xE1 0xA1 0x00 0x04]
int32 : [false, true]	[0xE2 0xA1 0x00 0xA1 0x01 0x04]
int32 : [5]	[0xE1 0xA2 0x05 0x00 0x00 0x00 0x04]
int32 : [10.0]	[0xE1 0xA3 0x00 0x00 0x00 0x00 0x00 0x24 0x40 0x04]

The grammar below defines four types of literals that are available for the metadata. Note that strings are represented indirectly using a 16 bit index into a string table.

```
MDT_BOOL = 0xA1;
MDT_INT32 = 0xA2;
MDT_FLOAT64 = 0xA3;
MDT_STRING = 0xA4;

false = 0x00;
true = 0x01;

literal =
  MDT_BOOL, (false | true) |
  MDT_INT32, int32 |
  MDT_FLOAT64, float64 |
  MDT_STRING, string_table_index;
```

For reasons of space an ellipsis (...) has been used in various places in the grammar. Lists with up to 32 elements are always represented using the byte codes `MDT_LIST_0, ..., MDT_LIST_31`. For example the empty list `[]` is designated with the byte value `MDT_LIST_0`. For lists with more than 31 elements it is necessary to parenthesise the elements with the `MDT_BEGIN_LIST` and `MDT_END_LIST` markers.

```
MDT_BEGIN_LIST = 0xA7;
MDT_END_LIST = 0xA8;

MDT_LIST_0 = 0xE0;
MDT_LIST_1 = 0xE1;
MDT_LIST_2 = 0xE2;
...
MDT_LIST_31 = 0xFF;

meta_data_list =
  MDT_LIST_0 |
  MDT_LIST_1, meta_data |
  MDT_LIST_2, meta_data, meta_data |
  ...
  MDT_LIST_31, meta_data, meta_data, ...,
      meta_data |
  MDT_BEGIN_LIST, {meta_data}, MDT_END_LIST;
```

Functors are similar to lists, except it is qualified with a 16 bit string table index, that designates the name of the functor. Note that identifiers without argument lists (like `hex` appearing in the example in section 6) use `MDT_FUNCTOR_0`.

```
MDT_BEGIN_FUNCTOR = 0xA5;
MDT_END_FUNCTOR = 0xA6;

MDT_FUNCTOR_0 = 0xC0;
MDT_FUNCTOR_1 = 0xC1;
MDT_FUNCTOR_2 = 0xC2;
...
MDT_FUNCTOR_31 = 0xDF;

meta_data_functor =
  MDT_FUNCTOR_0, string_table_index |
  MDT_FUNCTOR_1, string_table_index, meta_data |
  MDT_FUNCTOR_2, string_table_index, meta_data,
      meta_data |
  ...
  MDT_FUNCTOR_31, string_table_index, meta_data,
      meta_data, ..., meta_data |
  MDT_BEGIN_FUNCTOR, string_table_index,
      {meta_data}, MDT_END_FUNCTOR;
```

A metadata item is either a literal, list or functor. The overall metadata to be applied as a prefix to a type may optionally include a special 32 bit value representing up to 32 flags.

```
MDT_FLAGS = 0xA0;

meta_data =
  literal |
  meta_data_list |
  meta_data_functor;

flags = int32;

meta_data_qualifier =
  [MDT_FLAGS flags], meta_data_list;
```

9.2.5 Arrays

An array is specified by prefixing the element type by the byte code `FT_ARRAY` and the size of the array as a 32 bit integer.

C/C++ type	Byte Code
int32 x[3]	[0x27 0x03 0x00 0x00 0x00 0x04]
int32 * const x[3]	[0x27 0x03 0x00 0x00 0x00 0x1B 0x1C 0x04]

```
FT_ARRAY = 0x27;
array_size = int32;
array_type = FT_ARRAY, array_size, type;
```

9.2.6 Maps

A map with given key and value types is specified using the byte code `FT_MAP`

```
FT_MAP = 0x28;
key_type = type;
val_type = type;
map_type = FT_MAP, key_type, val_type;
```

9.2.7 Parameterised types

Consider a type that is parameterised by types named `T0, T1, ...` A byte code can support parameterised types (i.e. genericity) by using the following byte codes for the formal parameters:

```
FT_PARAM_0 = 0x40;  (* T0 *)
FT_PARAM_1 = 0x41;  (* T1 *)
FT_PARAM_2 = 0x42;  (* T2 *)
...
FT_PARAM_63 = 0x9F; (* T63 *)
```

Here are some examples of parameterised types:

C/C++ type	Byte Code
`const T0[3]`	`[0x27 0x03 0x00 0x00 0x00 0x1B 0x40]`
`map<T0,T1*>`	`[0x28 0x40 0x1C 0x41]`

9.2.8 Putting it all together

It is now a simple matter to collect together all the possible type specifiers:

```
type =
  basic_type |
  referenced_type |
  unary_type_qualifier type |
  array_type |
  map_type |
  meta_data_qualifier, type;
```

References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN-13: 9780201633610.

[2] Jens Palsberg and C. Barry Jay, *The Essence of the Visitor Pattern*, 1997 IEEE-CS COMPSAC.

[3] ISO/IEC 14977:1996(E). *Information technology - Syntactic metalanguage - Extended BNF*, First edition 1996-12-15. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.