SILICON VALLEY
CODE CAMP 2013

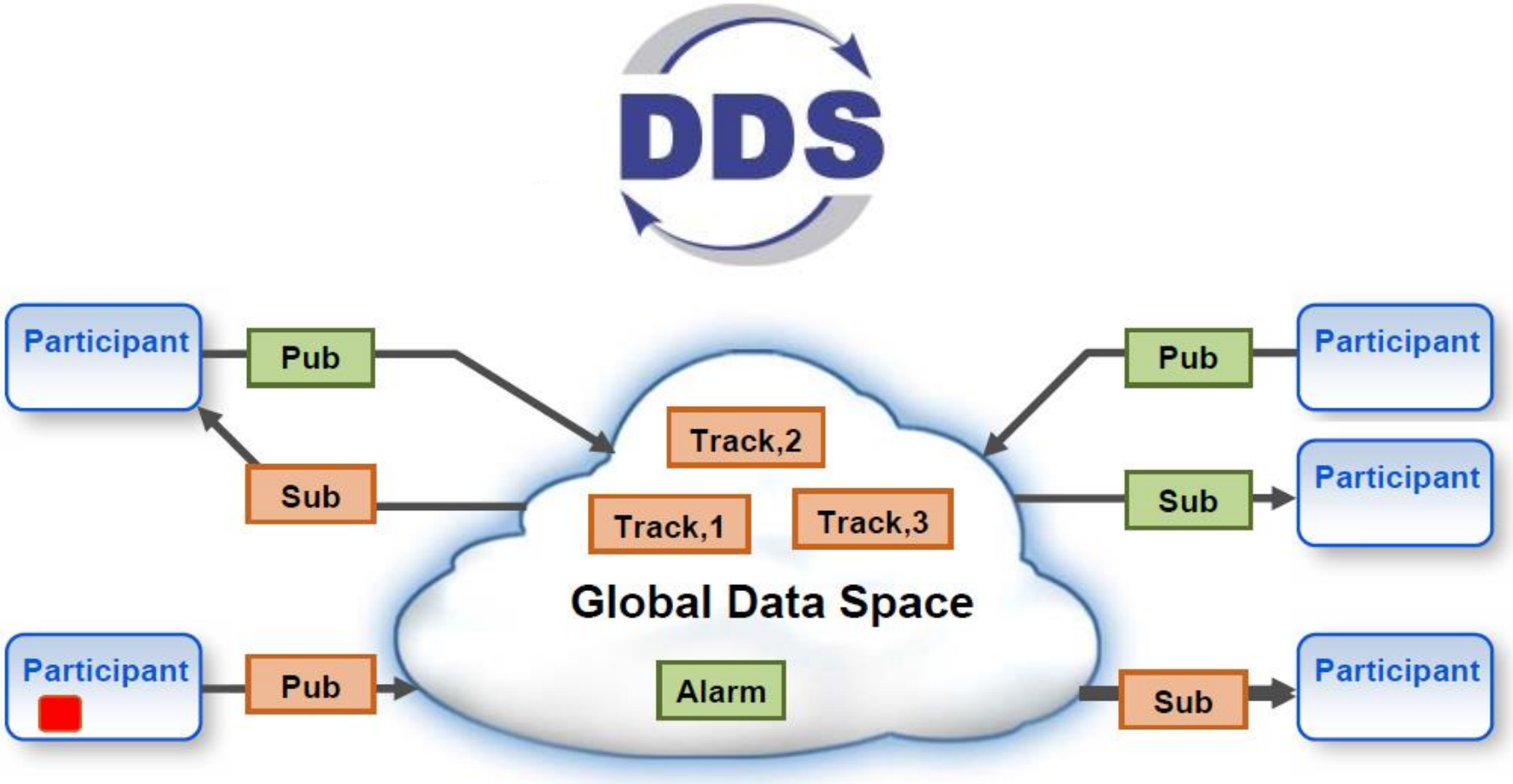
Your systems. Working as one.

Overloading in Overdrive: A Generic Data-Centric Messaging Library for DDS

Sumant Tambe, Ph.D.
Senior Software Research Engineer and Microsoft MVP
Real-Time Innovations, Inc.
www.rti.com

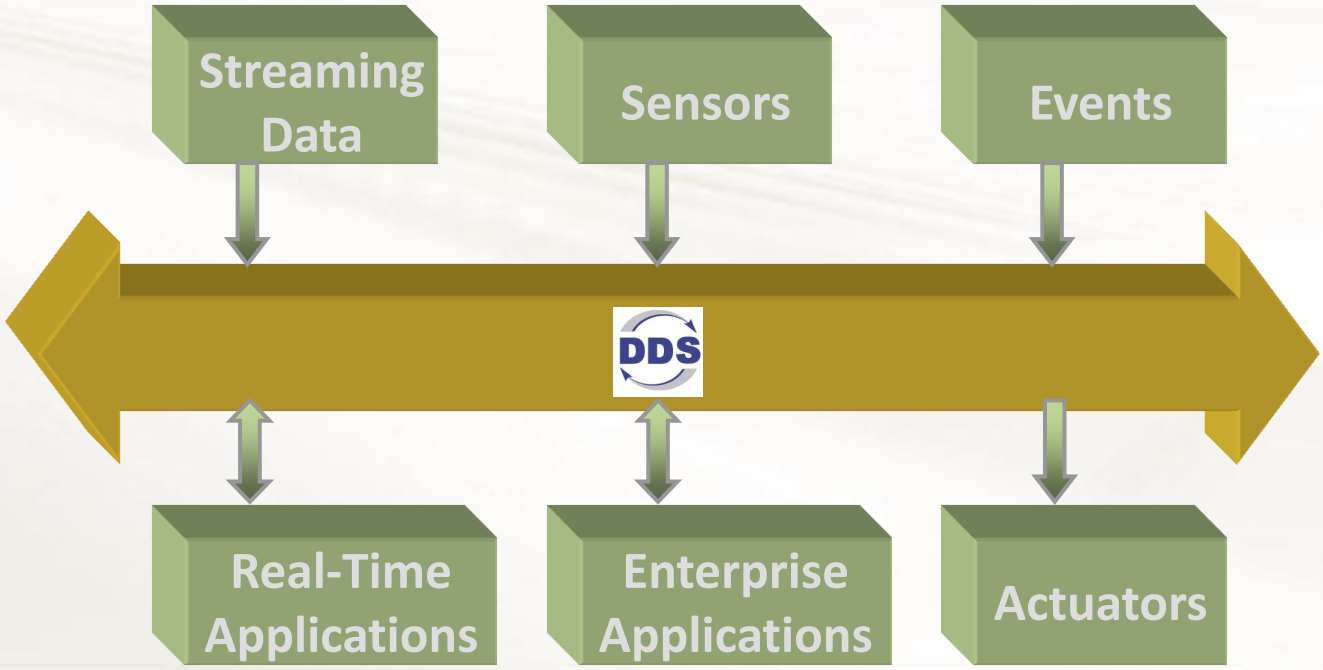
Data-Centric Communication Model in DDS

Provides a "**Global Data Space**" that is accessible to all interested applications.

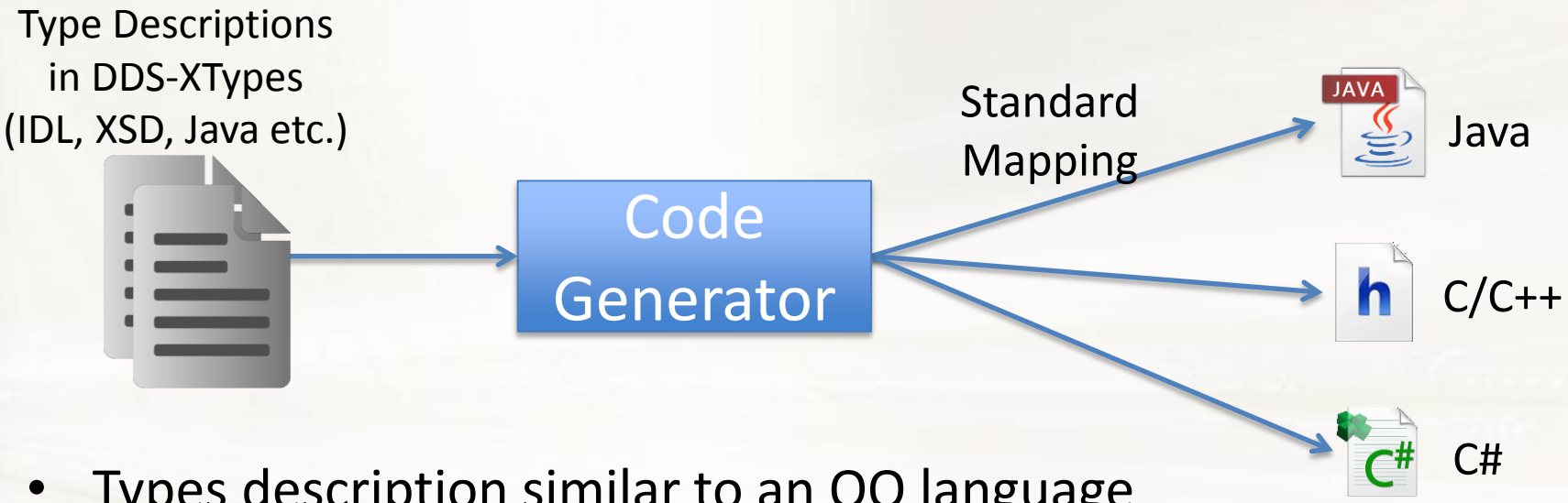


Data Distribution Service

- Data Distribution Service
  - Middleware knows the schema
  - Messages represent update to data-objects
  - Data-Objects identified by a key
  - Middleware maintains state of each object
  - Objects are cached. Applications can read at leisure
  - Smart QoS (Reliability, Ownership, History (per key), Deadline, and 18 more!)

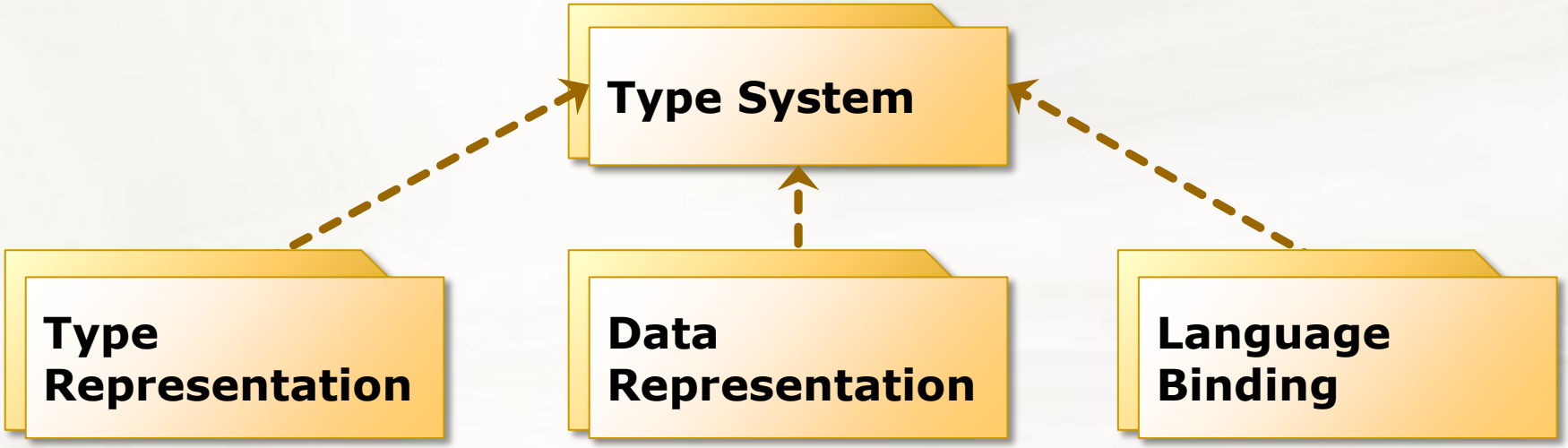


Data-Centric → Typed All the Way!



- Types description similar to an OO language
  - Primitives, strings, arrays and sequences, maps, structures, enumerations, aliases (typedefs), unions, modules, etc.
- Type versioning and assignability
- Meta-Data
  - Unique ID for each member
  - Annotations (like in Java)
  - Key/non-key
  - Optional/required members
  - Sharable/non-sharable

DDS X-Types Standard



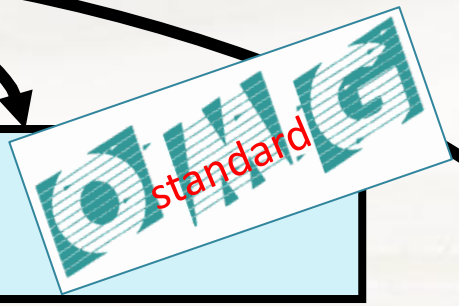
- **Type System**: DDS data objects have a type and obey the rules described herein
- **Language Binding for Object Manipulation**
  - Generate code when type-descriptions are available at compile-time
  - Otherwise, create/inspect objects of **any** type at run-time
- **Language Binding for Type Manipulation**
  - add/remove/query data members types (think Java reflection)
- **Data Representation**: Objects can be serialized for file storage and network transmission
- **Type Representation**: Types can be serialized for file storage and network transmission (using TypeObject)

Example

Type Representation

IDL: Foo.idl

```
struct Foo {
 string name;
 @key long ssn;
 @optional string
    phone;
};
```



Language Binding

IDL to C++ Language Mapping:

```
struct Foo {
  std::string name;
  int ssn;
  std::string * phone;
  // ...
};
```

Da Representation

IDL to CDR:

```
00000006
68656C6C
6F000000
00000002
```

Two Faces of DDS

- Application Integration
  - Many applications, many programming languages, many contractors necessitate a common format for sharing types
  - IDL serves well
  - Related technologies: CORBA Interfaces, schema definitions (DB), XML document structure (XSD)
- Messaging
  - Typically, a single (distributed) application
  - Likely developed using a single programming language
  - Overall system under single administration
  - Data already stored in application data structures
  - Just send it from point A to B
  - Related technologies: ActiveMQ, JMS
  - IDL not so much desirable

IDL may be a Distraction (for messaging apps)

- "… But I already have my headers…"
  - Maintaining redundant IDL type descriptions is a burden
  - Integrating generated code in app build system could be burdensome
  - Must write glue code
- Or sometimes it is an overhead
  - Copy data back and forth between app data types and IDL data types
- Application level data structures may not be captured accurately
  - Think XSD (e.g., sequence of choices)

Solution: A Pure C++11 Library-based Approach

Example:

```
// shape.h
struct ShapeType
{
  std::string color;
  int x;
  int y;
  unsigned shapesize;
};
```

```
// shape.h
RTI_ADAPT_STRUCT(
  ShapeType,
  (std::string, color, KEY)
  (int, x)
  (int, y)
  (unsigned, shapesize))
```


```
DDSDomainParticipant * participant = ...
GenericDataWriter<ShapeType> shapes_writer(participant, "Square");

for(;;) {
  ShapeType shape {“BLUE”, x, y, 30};
  shapes_writer.write(shape);
}
```

Solution: A Pure C++11 Library-based Approach

Example:

```
// shape.h
struct ShapeType
{
  std::string color;
  int x;
  int y;
  unsigned shapesize;
};
```

IDL

```
@Extensibility(EXTENSIBLE)
struct ShapeType {
  @key string color;
  long x;
  long y;
  unsigned long shapesize;
};
```

Instead

A substitute for lack of reflection in C++

```
// shape.h
RTI_ADAPT_STRUCT(
  ShapeType,
  (std::string, color, KEY)
  (int, x)
  (int, y)
  (unsigned, shapesize))
```

Read/Take Shapes (note C++11)

```
class MyShapesListener : public GenericDataReaderListener<ShapeType> {
public:
  void on_data_available(GenericDataReader<ShapeType> & dr) override {
    std::vector<ShapeType> samples = dr.take();
    for (auto const &shape : samples) {
      std::cout << “color = "     << shape.color     << “\n”
                << "x = "         << shape.x         << “\n”
                << "y = "         << shape.y         << “\n”
                << "shapesize = " << shape.shapesize << “\n”
                << std::endl;
    }
  }
};
```

```
DDSDomainParticipant * participant = ...
MyShapesListener listener;
std::string topic = “Square”;
GenericDataReader<ShapeType> shapes_reader(participant, &listener, topic);
for(;;)
  sleep(1);
```

Slightly Complex Example

```
struct VetoTDCHit  {
  int     hit_index;
  int     pmt_index; // not sent
  float   pmt_theta;
  float   pmt_phi;
};
typedef std::list<VetoTDCHit> VetoTDCHits;
struct VetoTruth
{
  int         sim_event;
  VetoTDCHits      hits;
};

enum STATUS_FLAGS { NORMAL=0,
                    ID_MISMATCH=1,
                    BAD_TIMESTAMP=2 };

struct EventInfo  {
  VetoTruth        truth;
  int           event_id;
  STATUS_FLAGS    status;
};
```

```
RTI_ADAPT_STRUCT(          // same header
  VetoTDCHit,
  (int,      hit_index)
  (float,    pmt_theta)
  (float,      pmt_phi))

RTI_ADAPT_STRUCT(
  VetoTruth,
  (int,         sim_event)
  (VetoTDCHits,     hits))

RTI_ADAPT_ENUM(
  STATUS_FLAGS,
  (NORMAL,         0)
  (ID_MISMATCH,    1)
  (BAD_TIMESTAMP,  2))

RTI_ADAPT_STRUCT(
  EventInfo,
  (VetoTruth,       truth)
  (int,           event_id,  KEY)
  (TATUS_FLAGS,   status))
```

Equivalent Type at Run-time

```
@Extensibility(EXTENSIBLE_EXTENSIBILITY)
enum STATUS_FLAGS {
     NORMAL = 0,
     ID_MISMATCH = 1,
     BAD_TIMESTAMP = 2,
};
@Extensibility(EXTENSIBLE_EXTENSIBILITY)
struct VetoTDCHit {
    long hit_index;
    float pmt_theta;
    float pmt_phi;
};
@Extensibility(EXTENSIBLE_EXTENSIBILITY)
struct VetoTruth {
    long sim_event;
    sequence<VetoTDCHit> hits;
};
@Extensibility(EXTENSIBLE_EXTENSIBILITY)
struct EventInfo {
    VetoTruth truth;
    @key long event_id;
    STATUS_FLAGS   status;
};
```

Type representation is binary at runtime but shown here using the IDL syntax for readability

So, How does it work?

- It uses:
  - C++11
  - Overloading in overdrive!
    - SFINAE: Substitution Failure Is Not An Error
  - Generic/Generative programming (templates)
  - Template meta-programming
  - Some (macro) preprocessor tricks
    - For code generation
  - C++ Standard Library
    - tuple, pair, vector, list, set, map, array, type_traits etc.
  - Boost
    - Fusion, Optional, Variant, and Preprocessor
  - TypeCode API
  - Dynamic Data API

What is Supported? (C++ point of view)

- Built-in types, arrays, enumerations
- Nested structs/classes (with public members)
- STL
  - string, vector, list, set, map, array, tuple, pair, iterators, etc.
- User-defined/custom containers
  - *my_special_container_with_iterators*
  - On-the-fly container adapters (views e.g., boost.Iterators)
- C++11 compilers
  - Visual Studio 2013 Preview
  - gcc 4.9
  - Clang 3.0, 3.2

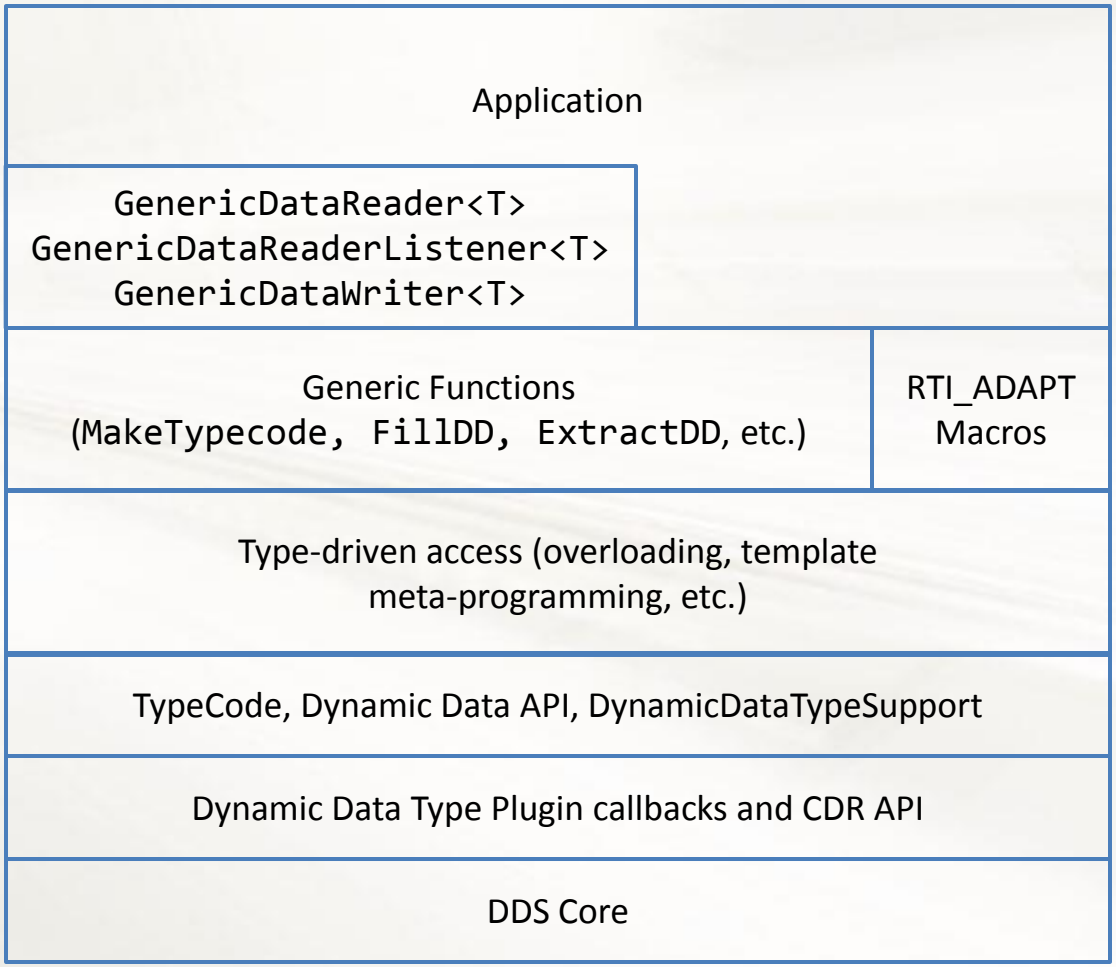
- In future…
  - Raw Pointers, smart pointers
  - Classes with public get/set methods?
  - Inheritance?

What is Supported? (IDL point of view)

- Basic types/enumerations/strings
- Arrays
- Sequences of strings/basic types/enumerations
- Bounded sequences/strings
- Structures
- Unions (including cases with defaults, multiple discriminators, and enumerations)
- Sparse Types
- Sequences of sequences (of sequences… and so on…)
- Sequences of structures
- Multidimensional arrays of strings, structures, sequences,…
- Nested structures
- Nested unions

- But, no bit-fields and inheritance, so far…

Architecture



Extremely Simplified DynamicData API

```
class DDS_TypeCode {};

class DDS_DynamicData
{
public:
  explicit DDS_DynamicData(const DDS_TypeCode &) {}

  void set_short     (int id, int16_t) {}
  void set_long      (int id, int32_t) {}
  void set_longlong  (int id, uint64_t) {}

  void set_octet     (int id, uint8_t) {}
  void set_ushort    (int id, uint16_t) {}
  void set_ulong     (int id, uint32_t) {}
  void set_ulonglong (int id, uint64_t) {}

  void set_float     (int id, float) {}
  void set_double    (int id, double) {}
  void set_longdouble(int id, long double) {}

  void set_string    (int id, const char *) {}
  void set_complex   (int id, const DDS_DynamicData &) {}
};
```

Overloading for Fundamental Types

```
void set_value(DDS_DynamicData & d, int id, bool v) {
   d.set_bool(id, v);      }

void set_value(DDS_DynamicData & d, int id, char v) {
   d.set_char(id, v);      }

void set_value(DDS_DynamicData & d, int id, int16_t v) {
   d.set_short(id, v);     }

void set_value(DDS_DynamicData & d, int id, int32_t v) {
   d.set_long(id, v);      }

void set_value(DDS_DynamicData & d, int id, float v) {
   d.set_float(id, v);     }

void set_value(DDS_DynamicData & d, int id, double v) {
   d.set_double(id, v);    }
```

Overloading set_value for std::string

```
void set_value(DDS_DynamicData & d,
               int id,
               const std::string & str)
{
   d.set_string(id, str.c_str());
}
```

Sequences
(simplified DDS DynamicData API, again)

```
class DDS_DynamicData
{
public:

  void set_boolean_seq    (int id, const DDS_BooleanSeq &) {}
  void set_long_seq       (int id, const DDS_LongSeq &) {}

  void set_float_seq      (int id, const DDS_FloatSeq &) {}
  void set_double_seq     (int id, const DDS_DoubleSeq &) {}
  void set_longdouble_seq (int id, const DDS_LongDoubleSeq &) {}
};
```

- Sequences are wrappers over C-style arrays
- Flexible: May own memory or loan/unloan from the user
  - RAII is used when memory is owned

Overloading for std::vector of primitives

```
void set_value(DDS_DynamicData & d,
               int id,
               const std::vector<float> & v)
{
   DDS_FloatSeq seq;
   seq.loan_contiguous(&v[0], v.size(), v.capacity());
   d.set_float_seq(id, seq);
   seq.unloan();
}

// and many many more similar looking functions for
// int8_t, int16_t, int32_t, int64_t, char, double, ...
```

set_value Template for std::vector of primitives

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const std::vector<T> & v)
{
    typename DynamicDataSeqTraits<T>::type seq;
    seq.loan_contiguous(&v[0], v.size(), v.capacity());

    typename DynamicDataSeqTraits<T>::SetSeqFuncPtr fn =
        DynamicDataSeqTraits<T>::set_seq_func_ptr;
    d.*fn(id, seq);
    seq.unloan();
}
```

Static method dispatch

- One template for all vectors of primitives
- Traits to get the function pointer
- Traits written using macros!
- Isn't that just overloading and static method dispatch? . . . Yes it is!

But set_value[T=bool] is outright wrong!

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const std::vector<T> & v)
{
   typename DynamicDataSeqTraits<T>::type seq;
   seq.loan_contiguous(&v[0], v.size(), v.capacity());

   typename DynamicDataSeqTraits<T>::SetSeqFuncPtr fn =
       DynamicDataSeqTraits<T>::set_seq_func_ptr;
   d.*fn(id, seq);
   seq.unloan();
}
```

std::vector<bool>
layout incompatible

- std::vector<bool> is often a space-efficient data structure
- Each element occupies a single bit instead of a single byte
- Does **not** follow many other std::vector invariants

Add std::vector<bool> overload of set_value

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const std::vector<T> & v) {
   typename DynamicDataSeqTraits<T>::type seq;
   seq.loan_contiguous(&v[0], v.size(), v.size());
   typename DynamicDataSeqTraits<T>::SetSeqFuncPtr fn =
       DynamicDataSeqTraits<T>::set_seq_func_ptr;
   d.*fn(id, seq);
   seq.unloan();
}
void set_value(DDS_DynamicData & d,
               int id,
               const std::vector<bool> & v)  {
   DDS_BooleanSeq seq;
   seq.ensure_length(v.size(), v.size());
   std::copy(begin(v), end(v), &seq[0]);
   d.set_boolean_seq(id, seq);
}
```

Overload for Classic Enumerations

- C++03 enumerations implicitly convert to an integer
  - User C++11's type-safe enums if you don't want that

- The following overload works ... for now!
- `set_value(DDS_DynamicData &, int id, int32_t)`

Overload for user-defined structs: “Foo”

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const T & t)
{
   DDS_DynamicData inner;

   // Use boost.fusion to iterate over the public members of T
   // and use set_value recursively to populate the inner object.
   // Effectively equivalent to

   set_value(inner, 0, at<0>(t));
   set_value(inner, 1, at<1>(t));
   set_value(inner, 2, at<2>(t)); // and so on...

   d.set_complex(id, inner);
}
```

Enumerations are user-defined types too!

- Addition of the template for Foo breaks enumerations

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const T & t);
```

All enums and user-defined structs resolve to this overload
(that's not what we want)

- Solution:
  - Function Overloading Based on Arbitrary Properties of Types
  - Powered by SFINAE!
    - **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror

A SFINAE Primer

- Instantiating a function template is a two-phase process
  - Deduce template arguments
  - Substitute the deduced type in all occurrences of the template parameter
- If the substitution process leads to an invalid type, argument deduction fails
- The function is removed from the overload resolution set

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type
negate(const F& f) { return -f(); }

negate(1);
```

Function Enablers and Disablers

- `enable_if<true>` keeps the template instantiation in the overload set
- `enable_if<false>` removes the template instantiation from the overload set

```
template <bool B, class T = void>
struct enable_if {
  typedef T type;
};
template <class T>
struct enable_if<false, T> {};

template <class T>
typename enable_if<is_arithmetic<T>::value, T>::type
foo(T t);

template <class T>
T bar (T t,
       typename enable_if<is_arithmetic<T>::value>::type* = 0);
```

Using enable_if with type_traits

- enable_if combined with type_traits is very powerful
- Many type_traits are implemented using compiler intrinsics
  - Think of it as compile-time reflection in library form
  - E.g., is_union, is_pod, is_enum, any many many more
  - C++11 type_traits library is portable

```
template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const T & t,
               typename enable_if<std::is_enum<T>::value>::type * = 0);

template <class T>
void set_value(DDS_DynamicData & d,
               int id,
               const T & t,
               typename enable_if<std::is_class<T>::value>::type * = 0);
```

Recap set_value overloads so far

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1
`template <class T>` `void set_value(DDS_DynamicData &, int, const std::vector<T> &); [T=fundamental]`	2
`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<std::is_enum<T>::value>::type * = 0);`
	4

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<std::is_class<T>::value>::type * = 0);`
	5

- Works with
  - 18 fundamental types, std::vector of those, std::vector<bool>, scalar enumerations, and scalar user-defined structs.
- No match for
  - std::vector<USER-DEFINED-STRUCTS>

More overloads

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`disable_if<is_fundamental<T>::value>::type * = 0);`
	3

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	4

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	6

- Works with
  - std::vector<USER-DEFINED-STRUCT>, std::vector<bool>, scalar enumerations, scalar user-defined structs, 18 fundamental types, std::vector of those
- Resolves incorrectly for std::vector<ENUMS>
  - Resolves to overload #3 but that's wrong because enums are not fundamental types

More overloads!

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`disable_if<is_fundamental<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

- std::vector<ENUMS> does not work due to ambiguity
- Overload #3 and #4 are both viable for std::vector<ENUMS>

More overloads!

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_class<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

- Support built-in arrays
  - Without array-specific overloads arrays map to the bool overload!

set_value Overloads for built-in arrays

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_class<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_fundamental<typename remove_all_extents<T>::type>::value>::type * = 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10

Overloads for std::vector<T[N]> ?

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_class<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_fundamental<typename remove_all_extents<T>::type>::value>::type * = 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10

Works for vector of structs but not for vector of built-in arrays

Overloads for std::vector<T[N]>

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`disable_if<is_fundamental<T>::value ||`
`is_enum<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool>`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_fundamental<typename remove_all_extents<T>::type>::value>::type * = 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10

Works for vector of structs/built-in arrays/nested vectors

The Overloads are Mutually Recursive

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`  enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`  disable_if<is_fundamental<T>::value ||`
`  is_enum<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`  enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

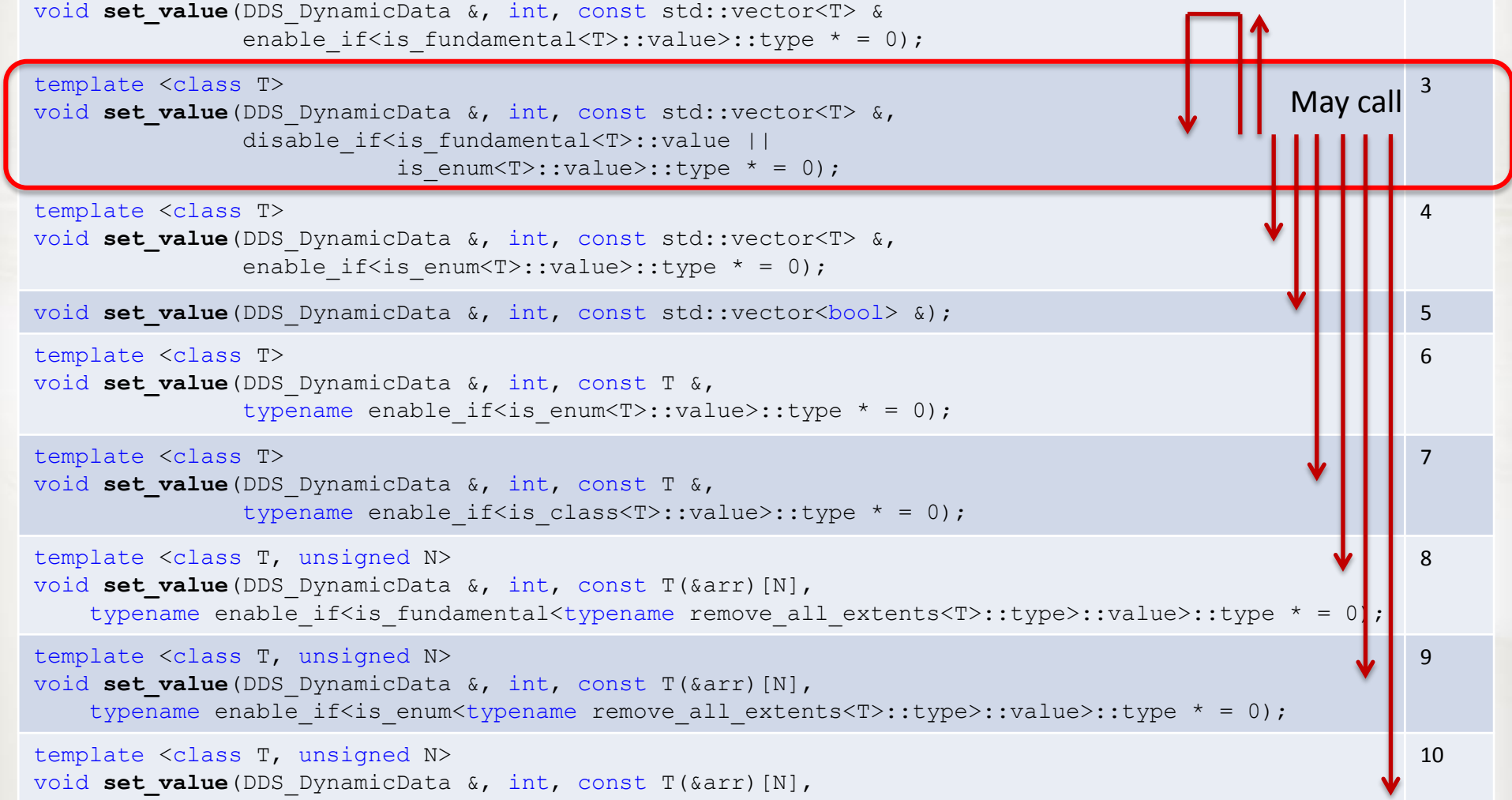
`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_fundamental<typename remove_all_extents<T>::type>::value>::type * = 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10



The Overloads are Mutually Recursive

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`  enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`  disable_if<is_fundamental<T>::value ||`
`  is_enum<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`  enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_fundamental<typename remove_all_extent...= 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`  typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10

Works for Multidimensional array or array of complex types

Overloading for Other STL Containers!

Overloads	#
`void set_value(DDS_DynamicData &, int, FundamentalTypes v); [18 overloads]`	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`enable_if<is_fundamental<T>::value>::type * = 0);`
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_class<T>::value>::type * = 0);`
	3

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &,`
`enable_if<is_enum<T>::value>::type * = 0);`
	4

`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_enum<T>::value>::type * = 0);`
	6

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`typename enable_if<is_class<T>::value>::type * = 0);`
	7

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_fundamental<typename remove_all_extents<T>::type>::value>::type * = 0);`
	8

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_enum<typename remove_all_extents<T>::type>::value>::type * = 0);`
	9

`template <class T, unsigned N>`
`void set_value(DDS_DynamicData &, int, const T(&arr)[N],`
`typename enable_if<is_class<typename remove_all_extents<T>::type>::value>::type * = 0);`
	10

What about standard list, set, deque, unordered, etc.?

Overloading for Other STL Containers!

```
template <class T>
struct is_container : std::false_type {};

template <class T, class Alloc>
struct is_container<std::vector<T, Alloc>> : std::true_type {};

template <class T, class Alloc>
struct is_container<std::list<T, Alloc>> : std::true_type {};

template <class T, class Comp, class Alloc>
struct is_container<std::set<T, Comp, Alloc>> : std::true_type {};

template <class Key, class Value, class Comp, class Alloc>
struct is_container<std::map<Key, Value, Comp, Alloc>> : std::true_type {};

// and few more ...
```

Overloading for Other STL Containers!

Overloads	#

`template <class T>`
`void set_value(DDS_DynamicData &, int, const std::vector<T> &`
`  enable_if<is_fundamental<T>::value>::type * = 0);` — vector<fundamental>
	1

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename disable_if<!is_container<T>::value ||`
`  std::is_fundamental<typename T::value_type>::value ||`
`  std::is_enum<typename T::value_type>::value >::type * = 0)` — T is a std container of complex type
	2

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_container<T>::value &&`
`  (is_fundamental<typename T::value_type>::value ||`
`  std::is_enum<typename T::value_type>::value) &&`
`  (is_vector<T>::value?`
`  is_bool_or_enum<typename T::value_type>::value :`
`  true)>::type * = 0);` — T is a std container of primitives but not vector <primitives except (bool & enums)>
	3

~~`void set_value(DDS_DynamicData &, int, const std::vector<bool> &);`~~	4

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<!is_container<T>::value &&`
`  is_class<T>::value>::type * = 0);` — Scalar complex type (Foo)
	5

`template <class T>`
`void set_value(DDS_DynamicData &, int, const T &,`
`  typename enable_if<is_enum<T>::value>::type * = 0);` — Scalar Enums
	6

Overloads for fundamental scalar types and built-in arrays are not shown

Using set_value overloads with Boost.Fusion

```
// shape.h
struct ShapeType
{
  std::string color;
  int x;
  int y;
  unsigned shapesize;
};
```

```
// shape.h
RTI_ADAPT_STRUCT(
  ShapeType,
  (std::string, color, KEY)
  (int, x)
  (int, y)
  (unsigned, shapesize))
```

```
struct SetValue {
    SetValue(DDS_DynamicData &dd);
    // ...
    template <typename T>
    void operator()(T& t) const {
      set_value(dd, ++id, t);
    }
};
ShapeType shape(“BLUE”, x, y, 30);
DDS_DynamicData dd(get_typecode<ShapeType>());
boost::fusion::for_each(shape, SetValue(dd));
```

More Info

- OMG DDS
  - http://portals.omg.org/dds/
- RTI Connext™ DDS
  - http://www.rti.com/products/dds/
- Example Code
  - https://cpptruths.googlecode.com/svn/trunk/cpp0x/overloading-overdrive.cpp

Your systems. Working as one.

Thank you!