FDM Data Marshalling Library

Randy Sargent
Anne Wright
Clay Kunz

Goals

- Provide a simple mechanism for marshalling/demarshalling objects in a variety of formats
- Support binary and text formats, tagged and untagged
  - Binary untagged (e.g. ACE CDR)
  - Text tagged (e.g. XML, Parse_Tree Text)
  - Binary tagged
- Support runtime object inspection
- Marshal existing types without modification
- Minimize burden on the objects themselves

Existing Mechanisms

- Binary: ACE CDR (Chris Urmson, user_curmson module)
- Human readable: Parse_Block (Clay Kunz, string_io module)
- Both work well in their domain, but supporting both would require 4 methods per data type (one input and output each)

Proposed Solution: FDM

- FDM stands for “Flexible Data Marshalling”
- One single method per object can both marshal and demarshal all supported formats
- Compatible with both CDR and Parse_Block
- Based on software developed by Randy Sargent outside NASA

Writing and reading CDR in memory

Writing objects to an ACE_Message_Block

```
FDM_CDR cdr_out(FDM::Write);
cdr_out.write(obj1);
cdr_out.write(obj2);
const ACE_Message_Block &mb=cdr_out.get_message_block();
```

Reading objects from an ACE_Message_Block

```
FDM_CDR cdr_in(mb, FDM::Read);
cdr_in.read(obj1);
cdr_in.read(obj2);
```

CDR File I/O

Writing objects to a file

```
FDM_CDR_File cdr_out("file.cdr", FDM::Write);
cdr_out.write(obj1);
cdr_out.write(obj2);
cdr_out.flush();
```

Reading objects from a file

```
FDM_CDR_File cdr_in("file.cdr", FDM::Read);
cdr_in.read(obj1);
cdr_in.read(obj2);
```

(warning, code on this page is untested)

Writing and reading Parse_Trees

- Parse_Tree is an in-memory, hierarchical representation for data

Writing an object to a Parse_Tree

```
1.    FDM_Parse_Tree fpt_out(FDM::Write);
2.    fpt_out.write(obj1);
3.    Parse_Tree pt= fpt_out.get_parse_tree();
```

OR

```
11.   Parse_Tree pt= FDM_Parse_Tree::object_to_parse_tree(obj1);
```

Reading an object from a Parse_Tree

```
21.   FDM_Parse_Tree fpt_in(pt, FDM::Read);
22.   fpt_in.read(obj1);
```

OR

```
31.   FDM_Parse_Tree::parse_tree_to_object(obj1, pt);
```

Converting objects to and from strings

- Parse_Tree Text is a text representation compatible with Parse_Block
- C-like syntax: **`{ foo1={ a=2; b=2.5; }; x=4.5; vec=[1 2 9 8]; }`**
- Pretty-printer outputs compact, properly indented structures

Writing an object to a string in Parse_Tree Text format

```
1.    FDM_Parse_Tree fpt_out(FDM::Write);
2.    fpt_out.write(obj1);
3.    Parse_Tree pt= fpt_out.get_parse_tree();
4.    std::string str = pt.pretty_print();
```

OR

```
11.   std::string str= FDM_Parse_Tree::object_to_string(obj1);
```

Reading an object from a string in Parse_Tree Text format

```
21.   pt.read_from_string(str);
22.   FDM_Parse_Tree fpt_in(pt, FDM::Read);
23.   fpt_in.read(obj1);
```

OR

```
31.   FDM_Parse_Tree::string_to_object(obj1, str);
```

Parse_Tree Text File I/O

Writing objects to an ostream in Parse_Tree Text format

```
1.    FDM_Parse_Tree::write_object(obj1, os);
2.    FDM_Parse_Tree::write_object(obj2, os);
```

Reading objects from an istream in Parse_Tree Text format

```
11.   FDM_Parse_Tree::read_object(obj1, is);
12.   FDM_Parse_Tree::read_object(obj2, is);
```

(warning, code on this page is untested)

Marshalling a simple class

- One way to marshal/demarshal a class is to write an “io” member function
- The “io” member function can both read and write
- Not necessary to keep separate input and output functions in sync!

- Example class Foo has two named fields:

```
class Foo {
public:
  int a;
  double b;

  bool io(FDM_Map map) {
    bool ok= true;
    ok &= map.field("a", a);
    ok &= map.field("b", b);
    return ok;
  }
};
```

Text: { a=3; b=9.5; }
CDR: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 23 40

Marshalling a simple class, contd.

- Example class Bar has three named fields, two of which are compound
- Marshalling compound field same as marshalling atomic field

```
class Bar {
public:
  Foo foo1;
  Foo foo2;
  double x;

  bool io(FDM_Map map) {
    bool ok= true;
    ok &= map.field("foo1", foo1);
    ok &= map.field("foo2", foo2);
    ok &= map.field("x", x);
    return ok;
  }
};
```

Text: { foo1={ a=2; b=2.5; }; foo2={ a=3; b=3.5; }; x=4.5; }

CDR: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 04 40 03 00 00 00 00 00 00 00 00 00 00 00 00 00 0C 40 00 00 00 00 00 00 12 40

Marshalling an array-like class

- Use FDM_Array if your object is more naturally represented as an array
- *This is an example only. Normally you would not subclass vector<T> solely to add an io method. See the next slide.*

```
template <class T>
class TestVector : public vector<T> {
public:
  bool io(FDM_Array array) {
    bool ok= true;

    // Read or write array length
    int len= size();
    ok &= array.resizable_length(len);

    // Resize the array based on the length.  When reading, this
    // is a no-op since the length doesn't change
    resize(len);

    for (int i= 0; i< len; i++)
      ok &= array.element((*this)[i]);
    return ok;
  }
};
```

Text: [11 22 33]
CDR: 03 00 00 00 6F 00 00 00 DE 00 00 00 4D 01 00 00

Marshalling std::vector<T>

- As an alternative to adding an “io” member function, a class can instead be made marshallable with an “io_object” function (*not* a member function)
- Write an “io_object” function to marshal third-party classes
- Like the “io” member function, “io_object” can both read and write an object’s data

```
template <class E>
bool io_object(FDM_Untyped_Node node, std::vector<E> &x) {
  FDM_Array array(node);
  bool ok=true;

  int size=x.size();
  ok &= array.resizable_length(size);
  x.resize(size);

  for (int i= 0; i< size; i++) ok &= array.element(x[i]);

  // The above loop could be replaced with the following
  // ok &= array.elements(x.begin(), x.end());

  return(ok);
}
```

Text: [11 22 33]
CDR: 03 00 00 00 6F 00 00 00 DE 00 00 00 4D 01 00 00

Marshalling C-style array field (wrong)

- What if you have a field which is a C-style array?
- map.field(array) doesn't work since the length information is lost

```
class ArrayField {
public:
  int x;
  int arr[10];

  bool io(FDM_Map map) {
    bool ok= true;
    map.field("x", x);
    map.field("arr", arr);  // this can't work!
  }
};
```

Marshalling C-style array field (correct)

- Correct solution for marshalling a C-style array field

```
class ArrayField {
public:
  int x;
  int arr[10];

  bool io(FDM_Map map) {
    bool ok= true;
    map.field("x", x);
    // Create an inline array field called "arr"
    FDM_Array a(map.field_node("arr"));
    a.elements(&arr[0], &arr[10]);
  }
};
```

Text: { x=100; arr=[0 1 2 3 4 5 6 7 8 9]; }

Four ways to marshal inline arrays

```
class ArrayFields {
public:
  int arr1[10];                          // Fixed-size array
  int arr2[10], arr2_nelems;             // Statically allocated, but only arr2_nelems are used
  int *arr3, arr3_nelems;                // Dynamically allocated, arr3_nelems elements long
  vector<int> arr4;                      // STL vector

  ArrayFields() { arr2_nelems=0; arr3= NULL; arr3_nelems= 0; }

  bool io(FDM_Map map) {
    bool ok= true;

    FDM_Array a1(map.field_node("arr1"));
    a1.elements(&arr1[0], &arr1[10]);

    FDM_Array a2(map.field_node("arr2"));
    a2.resizable_length(arr2_nelems);
    assert(arr2_nelems <= 10);           // Avoid buffer overrun on read
    a2.elements(&arr2[0], &arr2[arr2_nelems]);

    FDM_Array a3(map.field_node("arr3"));
    a3.resizable_length(arr3_nelems);
    arr3= (int*)realloc(arr3, arr3_nelems*sizeof(arr3[0]));
    a3.elements(&arr3[0], &arr3[arr3_nelems]);

    map.field("arr4", arr4);             // Marshalling the STL vector is easiest :-P
  }
};
```

Text: { arr1=[0 11 22 33 44 55 66 77 88 99]; arr2=[0 22 44]; arr3=[0 33 66 99 132]; arr4=[0 44 88 132 176 220 264]; }

Inheritance

- Subclass can call superclass **`io`** method.
- By convention, subclass calls superclass **`io`** before outputting subclass fields
- By making **`io`** virtual, we can output an object even if we don't know its subclass.

```
class Telem_Base {
public:
  Time t;
  virtual bool io(FDM_Map map) {
    bool ok= true;

    // "type" field used by factory
    string type_name= get_type_name();
    ok &= map.field("type", type_name);
    assert(type_name == get_type_name());

    ok &= map.field("time", t);
  }
  virtual string get_type_name() const { return "Telem_Base"; }
};

class Joint_Telem : public Telem_Base {
public:
  double joint_angle;
  double joint_velocity;
  virtual bool io(FDM_Map map) {
    bool ok= true;
    ok &= Telem_Base::io(map);
    ok &= map.field("joint_angle", joint_angle);
    ok &= map.field("joint_velocity", joint_velocity);
    return ok;
  }
  virtual string get_type_name() const { return "Joint_Telem"; }
};
```

Factories

- If you don't know the subclass of the object you're trying to read, you need to use a factory
- Very simple factory example:

```
// A real factory would probably want to make use of a runtime registration
// mechanism for subclass instantiation.

bool io_object(FDM_Map map, auto_ptr<Telem_Base> &telem) {
  bool ok= true;
  if (map.is_read()) {
    string type_name;
    ok &= map.peekfield("type", type_name);
    if (type_name == "Joint_Telem")     telem.reset(new Joint_Telem());
    else if (type_name == "Telem_Base") telem.reset(new Telem_Base());
    else return false; // unrecognized type!
  }
  ok &= telem->io(map);
  return ok;
}
```

Versioning and Magic Numbers

- Use a magic number to help check consistency
- Use a version to allow future code to read old data

```
// Class Baz, version 0
class Baz {
public:
  int a;
  double b;

  bool io(FDM_Map map) {
    bool ok= true;

    unsigned short magic = 0x26fa, version=0;
    ok &= map.field("magic", magic);
    ok &= map.field("version", version);
    assert(magic == 0x26fa);
    assert(version == 0);

    ok &= map.field("a", a);
    ok &= map.field("b", b);
    return ok;
  }
};
```

Versioning and Magic Numbers, contd.

- Version 1 can read data generated from a version 0 object, but will always write a version 1 object
- Version 0 cannot read a version 1 object (if written in an untagged format)
- We call this Microsoft-style backwards compatibility

```
// Class Baz, version 1
// Changes from version 0 shown in red
class Baz {
public:
  int a;
  double b;
  double c;

  bool io(FDM_Map map) {
    bool ok= true;

    unsigned short magic = 0x26fa, version=1;
    ok &= map.field("magic", magic);
    ok &= map.field("version", version);
    assert(magic == 0x26fa);
    assert(version <= 1);

    ok &= map.field("a", a);
    ok &= map.field("b", b);
    if (version >= 1) ok &= map.field("c", c);
    return ok;
  }
};
```

Implementing iostream << and >> with FDM

- The Parse_Tree Text pretty-printer makes decent-looking output, suitable in most cases for iostream <<

Example for class Foo:

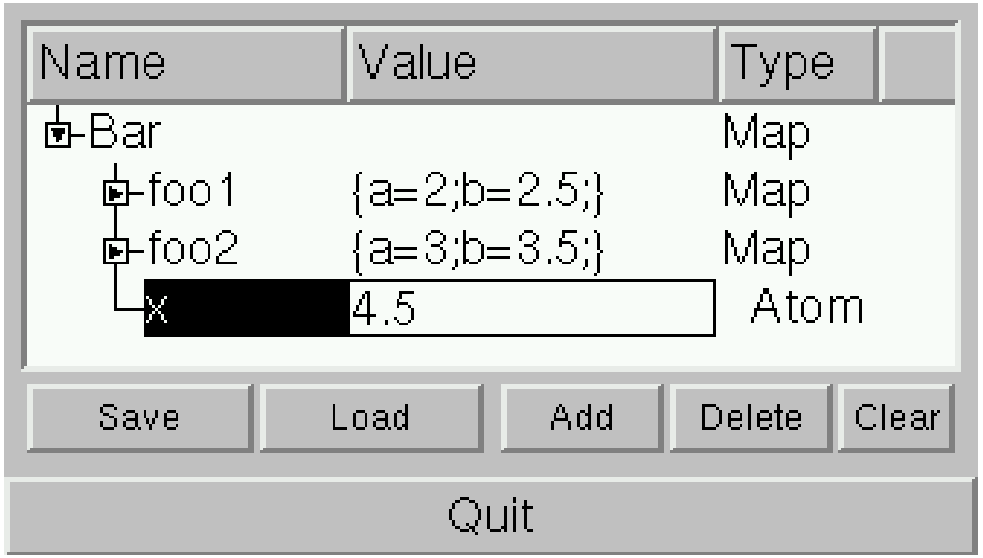
```
ostream& operator<<(ostream& os, const Foo& obj)
{
  FDM_Parse_Tree::write_object(obj, os);
  return(os);
}

istream& operator>>(istream& is, Foo& obj) {
  FDM_Parse_Tree::read_object(obj, is);
  return(is);
}
```

(warning, code on this page is untested)

Runtime object inspection

- Object inspector can display and modify any object which can marshal to FDM



Extending FDM to additional formats

- To support a new format, subclass FDM_Stream and define the following virtual member functions

```
// This is not a full definition, just the member functions which need to be overridden in subclasses
class FDM_Stream {
protected:
  virtual void _begin_array()                    =0;
  virtual bool _array_resizable_length(int &size)=0;
  // _array_element called before each element
  virtual bool _array_element()                  =0;

  virtual void _begin_map()                      =0;
  // _map_field called before each field
  virtual bool _map_field(const char *name)      =0;

  // _end ends either an array or map
  virtual void _end()                            =0;

  virtual bool _io_object(bool &x)               =0;
  virtual bool _io_object(unsigned char &x)      =0;
  virtual bool _io_object(char &x)               =0;
  virtual bool _io_object(unsigned short &x)     =0;
  virtual bool _io_object(short &x)              =0;
  virtual bool _io_object(unsigned int &x)       =0;
  virtual bool _io_object(int &x)                =0;
  virtual bool _io_object(unsigned long &x)      =0;
  virtual bool _io_object(long &x)               =0;
  virtual bool _io_object(unsigned long long &x) =0;
  virtual bool _io_object(long long &x)          =0;
  virtual bool _io_object(float &x)              =0;
  virtual bool _io_object(double &x)             =0;
  virtual bool _io_object(std::string &x)        =0;

  virtual bool _peek_object(unsigned int &x)     =0;
  virtual bool _peek_object(std::string &x)      =0;
}
```

Future work

- Add primitive type for binary chunk of data (currently represented as an array of **`unsigned char`**)
- Make **`pretty_print`** prettier, add level of detail limit
- Finish object inspector
- "Framed" map which would allow reading a subset of fields on an untagged stream. This would allow factories to instantiate an object's superclass if the correct subclass is not registered.
- Differentiate between errors due to
  - underlying stream (e.g. premature EOF or I/O error)
  - recoverable format inconsistency (asking for a non-existent field on a tagged stream)
  - unrecoverable format inconsistency (bad magic number on a non-tagged stream, or bad array size)
- Use exceptions?
- XML
- Tagged binary format
- Support for pointers to shared objects