Porting Your Software to libdap 3.5

James Gallagher*

2005-11-02
Revision: 12461

Contents

1 Use `BaseTypeFactory` instead of Virtual Constructors

In older versions of `libdap` (3.2 and earlier), the variables defined by your subclasses of `Byte`, . . . , `Grid` were created using 'virtual constructors'[1]. The code looked like:

Listing 1: Old Style Virtual Constructor

```
Byte *
NewByte(const string &n)
{
    return new NCByte(n);
}
```

In this example, the virtual constructor returns an instance of the `NCByte` class, which is apparently a subclass of `Byte` (although I don't show that explicitly, it's implied since the `NCByte` pointer is returned via a `Byte` pointer).

*OPeNDAP, Inc. jgallagher@opendap.org

The DDS parser knew to use those virtual constructors to make new instances of those types so that the instances would include the specializations you added.

This worked fine, why change it? Using the functions, only one set of specializations could be created per process. The virtual constructors have been replaced with a factory class. This enables one process to create many different set of specializations by making several different subclasses of the factory.

How it works in practice: The class `BaseTypeFactory` defines a factory class for the `Byte`, ..., `Grid` classes and provides a default implementation. To make a factory for your specializations, create a subclass of `BaseTypeFactory`. Here's an example:

Listing 2: New Factory Declaration

```
class NCTypeFactory: public BaseTypeFactory {
public:
    NCTypeFactory() {}
    virtual ~NCTypeFactory() {}

    virtual Byte *NewByte(const string &n = "") const;
    // more 'New' methods
};
```

And the implementation:

Listing 3: The Factory Implementation for NCByte

```
Byte *
NCTypeFactory::NewByte(const string &n ) const
{
    return new NCByte(n);
}
```

1.1 Using the factory

It's great to have the factories, but how does the DDS class know to use it? It now has a constructor that takes a pointer to a `BaseTypeFactory` as a formal parameter. This determines which factory will be used to instantiate variables by `DDS::parse()`. It's the caller's responsibility to free the factory once the DDS instance is deleted. See the documentation for those classes for more information about the new constructors.

Listing 4: DDS Constructor

```
DDS(BaseTypeFactory *factory, const string &n = "");
```

See Appendix A for the complete declaration and definition files.

2 STL Iterators Versus `Pix`

Thanks to the work of Patrick West, `libdap` now has a complete set of STL iterators that act as replacements for the aging `Pix` iterators. There are new methods and types for all the classes that act as containers (DDS,

DAS, Constructor, AttrTable). Here's an example of the old Pix code used to access things in an Structure variable:[1]

Listing 5: Pix iterators

```
Structure *structure;
BaseType *var;
Pix p = structure->first_var();
while (p) {
  var = structure->var(p);
  // do stuff
  structure->next_var(p);
}
```

Here's the code rewritten using the STL iterators:

Listing 6: STL iterators

```
BaseType *var;
Constructor::Vars_iter p = structure->var_begin();
while (p != structure->var_end()) {
  var = *p;
  // do stuff
  ++p;
}
```

Some things to note:

- Each of the container types has a method similar to STL's `begin()` method which returns an iterator that references the start of the contained items.
- Each of iterators 'points to a pointer.' To get the `BaseType` pointer that the iterator `p` references, use the dereferencing operator `*` (*i.e.* `var = *p`; note that `var` is a pointer to a `BaseType`).
- To advance to the next item, increment the iterator.
- To test for the end of the container, use the `var_end()` method. You can assign the value of `var_end()` to a variable and test against that if you think the cost of calling the method in a loop is too high.

Note that the iterators used for Structure and Sequence are defined in their parent class `Constructor` while `Grid`, `DDS`, `DAS` and `AttrTable` contain their own type definitions. Look at the class' documentation to see which new methods replace the '`Pix` methods.'

Note that Patrick also created a set of `Pix` adapters that enable the old Pix types to work by adapting them to STL iterators. It's best to treat all the `Pix` code as deprecated, but the adapters ease the work by enabling you to break it up over several releases.

[1] Example code from the gadods library written by Joe Wielgosz at COLA.

3 New Methods in Connect

Connect no longer contains instances[2] of DAS, DDS, *et c.*, that can hold values returned from the server. Instead it has methods which return the objects using value-result parameters. Here are the new methods:

Listing 7: Methods from Connect

```
1   virtual void request_das(DAS &das) throw(Error, InternalErr);
    virtual void request_das_url(DAS &das) throw(Error, InternalErr);
3
    virtual void request_dds(DDS &dds, string expr = "") throw(Error, InternalErr);
5   virtual void request_dds_url(DDS &dds) throw(Error, InternalErr);

7   virtual void request_data(DataDDS &data, string expr = "") throw(Error, InternalErr)
    virtual void request_data_url(DataDDS &data) throw(Error, InternalErr);
```

Note:

- The methods now take a reference to the object being requested; the return is via this parameter.
- The methods signal errors using exceptions.
- Patrick has added versions that do not modify the URL at all (the ones with the `_url` suffix).

Other changes: Be sure to scan the `Connect` documentation for other things that have been deprecated. The Connect class was one of the poorest in `libdap`. I have reduced it considerably and factored many of the HTTP-specific methods into `HTTPConnect` and all of the caching into `HTTPCache`. Also note that cache resource management is not controlled by the `Response`, `HTTPResponse` and `HTTPCacheResponse` classes. Particularly, the release of cache resources is now managed by the `Response` class and its children.

3.1 Connections removed

In the past `libdap` included a class that could be used to make a collection of `Connect` instances. This was used in code like the NetCDF Client Library to store one instance of `Connect` for each 'open file.' That class was called `Connections`. It has been removed from the library since it makes most sense to just put a copy of it in your project. If you need it, download the NetCDF client library source code (or use subversion) and put copies of `Connections.cc` and `Connections.h` in your source.

4 Automake support

All of OPeNDAP's projects will use autoconf, automake and libtool to build on Unix platforms. To make it easier to write configure.ac scripts which detect libdap, we include an autoconf m4 macro. Look in the `m4` directory of libdap for the file `libdap.m4`. Copy this into your source. Here's an example of the macro's use:[3]

[2] Well, they are actually still there, but the methods that access them are deprecated and will be removed in 3.6.

[3] Thanks to Patrice Dumas for contributing the macro and many patches for our builds.

Listing 8: libdap.m4

```
AC_CHECK_LIBDAP([3.5.0],
  [
   LIBS="$LIBS $DAP_LIBS"
   CPPFLAGS="$CPPFLAGS $DAP_CFLAGS"
  ],
  [ AC_MSG_ERROR([Cannot find libdap])
 ])
```

A NCTypeFactory

Listing 9: NCTypeFactory.h

```
// -*- mode: c++; c-basic-offset:4 -*-

// This file is part of libdap, A C++ implementation of the OPeNDAP Data
// Access Protocol.

// Copyright (c) 2005 OPeNDAP, Inc.
// Author: James Gallagher <jgallagher@opendap.org>
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// You can contact OPeNDAP, Inc. at PO Box 112, Saunderstown, RI. 02874-0112.

#ifndef nc_type_factory_h
#define nc_type_factory_h

#include <string>

// Class declarations; Make sure to include the corresponding headers in the
// implementation file.
```

```
33
  class NCByte;
35 class NCInt16;
  class NCUInt16;
37 class NCInt32;
  class NCUInt32;
39 class NCFloat32;
  class NCFloat64;
41 class NCStr;
  class NCUrl;
43 class NCArray;
  class NCStructure;
45 class NCSequence;
  class NCGrid;
47
  /** A factory for the NetCDF client library types.
49
      @author James Gallagher
51    @see DDS */
  class NCTypeFactory: public BaseTypeFactory {
53 public:
      NCTypeFactory() {}
55    virtual ~NCTypeFactory() {}

57    virtual Byte *NewByte(const string &n = "") const;
      virtual Int16 *NewInt16(const string &n = "") const;
59    virtual UInt16 *NewUInt16(const string &n = "") const;
      virtual Int32 *NewInt32(const string &n = "") const;
61    virtual UInt32 *NewUInt32(const string &n = "") const;
      virtual Float32 *NewFloat32(const string &n = "") const;
63    virtual Float64 *NewFloat64(const string &n = "") const;

65    virtual Str *NewStr(const string &n = "") const;
      virtual Url *NewUrl(const string &n = "") const;
67
      virtual Array *NewArray(const string &n = "", BaseType *v = 0) const;
69    virtual Structure *NewStructure(const string &n = "") const;
      virtual Sequence *NewSequence(const string &n = "") const;
71    virtual Grid *NewGrid(const string &n = "") const;
  };
73
  #endif // nc_type_factory_h
```

Listing 10: NCTypefactory.cc

```
2 // -*- mode: c++; c-basic-offset:4 -*-

4 // This file is part of libdap, A C++ implementation of the OPeNDAP Data
```

```
// Access Protocol.

// Copyright (c) 2005 OPeNDAP, Inc.
// Author: James Gallagher <jgallagher@opendap.org>
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// You can contact OPeNDAP, Inc. at PO Box 112, Saunderstown, RI. 02874-0112.


#include <string>

#include "NCByte.h"
#include "NCInt16.h"
#include "NCUInt16.h"
#include "NCInt32.h"
#include "NCUInt32.h"
#include "NCFloat32.h"
#include "NCFloat64.h"
#include "NCStr.h"
#include "NCUrl.h"
#include "NCArray.h"
#include "NCStructure.h"
#include "NCSequence.h"
#include "NCGrid.h"

#include "NCTypeFactory.h"
#include "debug.h"

Byte *
NCTypeFactory::NewByte(const string &n ) const
{
    return new NCByte(n);
}

Int16 *
```

```
NCTypeFactory::NewInt16(const string &n ) const
{
    return new NCInt16(n);
}

UInt16 *
NCTypeFactory::NewUInt16(const string &n ) const
{
    return new NCUInt16(n);
}

Int32 *
NCTypeFactory::NewInt32(const string &n ) const
{
    return new NCInt32(n);
}

UInt32 *
NCTypeFactory::NewUInt32(const string &n ) const
{
    return new NCUInt32(n);
}

Float32 *
NCTypeFactory::NewFloat32(const string &n ) const
{
    return new NCFloat32(n);
}

Float64 *
NCTypeFactory::NewFloat64(const string &n ) const
{
    return new NCFloat64(n);
}

Str *
NCTypeFactory::NewStr(const string &n ) const
{
    return new NCStr(n);
}

Url *
NCTypeFactory::NewUrl(const string &n ) const
{
    return new NCUrl(n);
}

Array *
```

```
NCTypeFactory::NewArray(const string &n , BaseType *v) const
{
    return new NCArray(n, v);
}

Structure *
NCTypeFactory::NewStructure(const string &n ) const
{
    return new NCStructure(n);
}

Sequence *
NCTypeFactory::NewSequence(const string &n ) const
{
    return new NCSequence(n);
}

Grid *
NCTypeFactory::NewGrid(const string &n ) const
{
    return new NCGrid(n);
}
```

References

[1] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.