Implementing a C++ interface for netCDF-4

Shanna-Shaye Forbes

Academic Affiliation, Fall 2007: Graduate Student, The University of California at Berkeley
shannashaye@berkeley.edu

SOARS® Summer 2007

Science Research Mentor: Ed Hartnett
Writing and Communication Mentor: Richard Loft

ABSTRACT

The network common data form (netCDF) was created by Unidata at the University Corporation for Atmospheric Research to simplify data access and sharing in the atmospheric and geoscience science community. Even though the current full release of netCDF, known as netCDF-3, has proven to be successful, increasing data complexity and user demands for increased functionality have necessitated a new release of netCDF with improved functionality and the ability to store user defined data types. With these requirements in mind, Unidata created netCDF-4, which augments the functionality of netCDF-3 with more flexible ways to add data, better support for custom data structures, and simplified use of another more complex data model, known as HDF5. At present netCDF-4 interfaces exist for C and Fortran, however none exists for C++. This project’s aim was to design and implement a C++ interface for netCDF-4. The netCDF-4 C++ interface was implemented as a thin layer on top of the netCDF-4 C interface; its design, however, allows all the functionality of a full C++ implementation. The netCDF-4 C++ interface will allow data providers and developers with a preference for C++ to take advantage of the new features netCDF-4 offers for creating portable, self-describing datasets. A prototype version of the C++ interface is available for evaluation and testing from http://www.unidata.ucar.edu/software/netcdf/netcdf-4. Comments should be directed to Ed Hartnett at ed@unidata.ucar.edu.

The Significant Opportunities in Atmospheric Research and Science (SOARS) Program is managed by the University Corporation for Atmospheric Research (UCAR) with support from participating universities. SOARS is funded by the National Science Foundation, the National Oceanic and Atmospheric Administration (NOAA) Climate Program Office, the NOAA Oceans and Human Health Initiative, the Center for Multi-Scale Modeling of Atmospheric Processes at Colorado State University, and the Cooperative Institute for Research in Environmental Sciences. SOARS also receives funding from the National Center for Atmospheric Research (NCAR). SOARS is a partner project with Research Experience in Solid Earth Science for Student (RESESS).
Section 1. INTRODUCTION

For many years no standard data model or data format was available for the atmospheric science or geoscience community. As a result, data users had to have knowledge of the many different formats used by data providers. In addition, users were required to have knowledge of, or worse, develop format-specific data analysis or visualization tools. This burden on users significantly hindered the dissemination of data within the atmospheric and geoscience community.

One development to simplify the interchange problems caused by numerous data formats was netCDF, and another was HDF. In the late 1980s, the Unidata Program Center at the University Corporation for Atmospheric Research produced the Network Common Data Form (netCDF) and the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign created the Hierarchical Data Format (HDF). HDF was developed specifically for high performance computing and netCDF for more general use. Both data formats were developed independently, until it became obvious that netCDF users could benefit significantly from some of the basic features of the HDF format. The netCDF data model was then revised to incorporate aspects of the HDF data model.

Over the years, scientific data providers and developers of scientific data access tools have made extensive use of the netCDF data model and data format in archives and tools to suit the needs of their users. The development of netCDF has had a significant impact on the scientific data community, and has led to the development of a variety of application programming interfaces, the boundary between a reusable library of software and the applications that make use of it.

Increasingly, object oriented middleware and tools are being developed for the scientific data community, and as a result there is a need for an object oriented interface to netCDF. This project implements such an interface in C++ for netCDF-4. This interface promotes desirable object oriented design features such as data encapsulation and data manipulation.

This paper is divided into five sections. Section 1 serves as an introduction to the project, section 2 provides background information about netCDF and HDF, section 3 includes a discussion of the interface design and an overview of the C++ interface, section 4 presents a summary of the impacts of the newly created interface, and section 5 contains a discussion of possible future developments.

Section 2. BACKGROUND

This section discusses netCDF, HDF5, and netCDF-4.

2.1 The Network Common Data Form (netCDF)

NetCDF began as an attempt to provide a machine independent interface between data providers and data users. NetCDF was first implemented at Unidata in 1988 in collaboration with Joe Fahle at SeaScape, Michael Gough at Apple, and Angel Li at the University of Miami.
NetCDF initially had interfaces for C and Fortran, and a C++ interface was added in 1993. The netCDF release was later modified to include companion utilities ncdump and ncgen [1].

NetCDF is a data model and data format for array based scientific data that is widely used in the atmospheric science and geoscience community. In addition to being machine independent, netCDF allows appending of data without copying the data set or redefining its structure [2].

![The NetCDF-3 Data Model](image)

Figure 1. UML Diagram of the netCDF-3 data model

In the netCDF data model, shown in Figure 1, a File generally consists of three kinds of objects: Attributes, Variables, and Dimensions. Attributes can either apply to a File or to a Variable. File attributes contain information about all the data in a file, and Variable attributes describe the contents of each variable. While Variable objects store data, Dimension objects are used to describe the shape of variables. Variables with the same Dimensions share the same grid. Variable types are one of six primitive types: char, byte, short, int, float, or double. An example of a netCDF file's metadata generated with the netCDF utility ncdump is shown in Figure 2.
In the netCDF-3 release, a netCDF File could contain any number of Dimensions but only one unlimited Dimension. Unlimited dimensions are used with model or observational data and represent a growing Dimension. For example, a three dimensional variable temperature could be created based on latitude, longitude, and time. While latitude and longitude are normal

```netcdf
sfc_pres_tmp {
  dimensions
    Latitude = 68;
    Longitude = 128;
  variables:
    float latitude(Latitude);
    latitude units = 'degrees north';
    float longitude(Longitude);
    longitude units = 'degrees east';
    float pressure(latitude, longitude);
    pressure units = 'hPa';
    float temperature(latitude, longitude);
    temperature units = 'celsius';
  data:
    Latitude = 25, 30, 35, 40, 45, 50;
    Longitude = -125, -120, -115, -110, -105, -100, -95, -90, -85, -80, -75, -70;
    pressure = 980, 936, 918, 924, 930, 936, 942, 948, 954, 960, 966,
               972, 978, 984, 990, 996, 1002, 1008, 1014, 1020, 1026;  
    temperature = 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27;
  }
```

**Figure 2. Metadata for a netCDF file generated with ncdump showing two dimensional pressure and temperature variables**

“fixed” dimensions, an unlimited dimension is desirable for the time dimension. This allows appending temperature readings to data without removing the current variable and replacing it with a new variable containing all the updated data.

The current full release of netCDF, netCDF-3, includes a library written in ANSI C as well as an improved Fortran interface. Although netCDF-3 included many improvements over netCDF-2, the interface is limited by a small number of external numeric data types and restrictions on data complexity. The netCDF data model, shown in Figure 1, is limited because it does not support nested data structures such as trees, nested arrays, or other recursive structures [2].

### 2.2 Hierarchical Data Format (HDF5)

In this section, we review the aspects of HDF5 relevant to the netCDF-4 release. HDF5 has been primarily funded by NASA and the Department of Energy (DoE) for high-performance computing. It supports parallel I/O, and it has a richer and more complex data model than netCDF. Parallel I/O is input and output from a program broken into multiple parts and running simultaneously on multiple CPUs.
Figure 3. UML Diagram for the HDF5 data model

A simplified version of the HDF5 data model is shown in Figure 3. HDF5 Datasets/Files are organized in a hierarchical structure and support a tree structure collection of items similar to a UNIX style file directory system. Datasets consist of Groups of data with similarities or data collected in the same region. Groups consist of subgroups, Variables which store data, as well as Attributes which describes the contents of Variables and information in a Group. HDF5 Variables can be one of many primitive types or seven user defined types. HDF also supports large files, parallel I/O, and allows arrays to consist of compound data structures, and user defined types. Other features of the HDF5 data model will be discussed in the next section on netCDF-4 since some of netCDF-4’s features are merely simplifications of the HDF5 data model.

2.3 The Network Common Data Form-4 (netCDF-4)

NetCDF-4 features improvements on netCDF-3 as well integration with the HDF5 data model. The basic aspects of the HDF5 data model included in netCDF-4 include groups, multiple unlimited dimensions, more primitive data types, four user defined data types (enum, opaque, compound, and variable length). Storage of several values of possibly different types in an array is supported through the use of the compound data type.

The netCDF-4 data model is presented in Figure 4 and items in red are additions to the netCDF-3 data model. The netCDF-3 primitive types were expanded in netCDF-4 to include the primitive types shown in red in figure 4 and the user-defined types enum, opaque, compound, and variable length mentioned in the paragraph above were adopted from the HDF5 data model.
Each netCDF-4 File consists of a default root Group, and each Group whether the root group or a user created group can consist of Attributes, Dimensions, Variables, and other subgroups, as well as User Defined Types. Attributes can now either be of a primitive type or a user defined type, and a Group can contain Variables with multiple unlimited Dimensions. In addition a netCDF-4 variable can either be a primitive type or a user-defined type. The User Defined Types allowed for Attributes or Variables are Enum, Opaque, Compound, and VariableLength types. The Enum type is a representation of the enumeration type, and the Opaque type is used to store a named fixed-size sequence of bytes which are stored as raw encoded or untyped data blocks. The Compound type is used to store data of multiple types in a structure, and the VariableLength type is used for the storage of varying length arrays.

A file has a top-level unnamed group. Each group may contain one or more named subgroups, user-defined types, variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One or more dimensions may be of unlimited length.

Figure 4. UML Diagram of the netCDF-4 data model
Section 3. METHODOLOGY AND DESIGN

Before beginning the netCDF-4 C++ interface, we needed to determine which aspects of the netCDF-3 C++ interface would be useful in netCDF-4, what other characteristics were necessary to make the interface easy to use, and also how to incorporate the desired aspects of HDF5. To determine the desired aspects of the interface, we sorted through netCDF C++ suggestions and requests submitted to the netCDF newsgroup. We then examined a netCDF-3 C++ wrapper library created by Fei Lui at NOAA [Appendix A], and subsequently performed a critical analysis of aspects of the C++ standard template library which would prove useful to the latest interface. The requirements that emerged from this analysis were the need for generic programming implementation, and the advantages provided by powerful C++ features such as templates, a single namespace, iterators, and exceptions. Templates help to reduce redundancy in code and help to streamline method testing. Namespaces help decrease collision conflicts between multiple classes. Iterators enable efficient traversal between objects, and exceptions make error handling more efficient.

3.1 Design Methodology

3.1.1 Pre-Interface Design

Our design goals were influenced by Kenneth Arnold’s Article “Programmers are people too” in which he discusses design concepts such as progressive disclosure, simple and intuitive API development, as well as well tested APIs [5]. Based on these ideas we ultimately decided that our interface needed to be an extension of the functionality of the netCDF-3 C++ interface, it needed to be intuitive and easy to use, and needed to be extensively tested.

Our goal to have the netCDF-4 C++ interface serve as an extension of the functionality of the previous interface was a result of Arnold’s discussion of progressive disclosure. Progressive disclosure divides tasks or information into beginning, intermediate, and expert stages. This division ensures that the beginning user is not overwhelmed by all the technical details that an expert would welcome. Arnold compares progressive disclosure to driving a car. The beginning user only needs to know how to drive the car, while engine work requires the intervention of an expert [5]. We ultimately chose to adopt this as a goal because we wanted the user to see using the new interface as easy if not easier to use than the previous interface.

Our second goal was to have a simple yet intuitively designed interface. We wanted the interface to conform to standard practices of the data community. We opted for this route because Arnold suggested that when designing a good user interface one should keep in mind who you are designing for and what they expect the interface to do. This approach enables the designer to anticipate what the user will expect from a method without requiring that the user read all the documentation. Arnold points out a simple but important fact that few if any programmers read all of the provided documentation. With this knowledge in hand the designer must simply be aware that an inconsistently designed system is hard to use and hard to fix [5].

Our final goal was to have an interface that was very well tested. Though this goal is standard practice with all Unidata releases, we felt it was an important goal to emphasize because, as
Arnold notes “programmers are not only human but also notoriously lazy problem solving humans.” Even if you try to enforce traditional programming conventions to ensure that no errors are made, programmers will generally combine actions or will find workarounds. Arnold states that it is also important to remember that, while you may have become an expert after designing your interface, keep in mind as a designer that the user just “wants everything to work”[5].

3.1.2 Interface Design

To address user requirements three possible implementations of the netCDF-4 library were considered. The first option was to implement the netCDF-4 interface entirely on top of the current netCDF-4 C interface, the second option to implement it on top of the netCDF-3 C++ interface and the netCDF-4 C interface, and the third option to implement it entirely on top of the HDF-5 interface. Implementing directly on top of the HDF5 C interface would decrease the number of layers in the new C++ API implementation, however, this option was not chosen because backward compatibility would not have been possible. We selected the first option to implement the new C++ interface as a thin layer on top of the current netCDF-4 C interface, because an extensive set of tests exists for the netCDF-4 C interface but not for the netCDF-3 C++ interface, and also because this option enables backward compatibility.

![NetCDF-4 Software Architecture](image)

*Figure 5. NetCDF-4 Software Architecture selected for the C++ API*

The software architecture diagram shown in Figure 5 is a visual representation of the layers each version of netCDF uses. At the third level from the bottom in Figure 5, the HDF5 C API is extended to the left to show that the HDF5 C API uses both POSIX and MPI I/O. Instead of creating the entire C++ interface from scratch, useful aspects of the netCDF-3 interface were adopted. The classes in the netCDF-4 C++ interface are shown in Figure 6. Classes in blue
represent original classes that have been modified to suit the needs of netCDF-4 and classes in red were created from scratch during this project.

![Diagram of netCDF-4 C++ classes](image)

*Figure 6. netCDF-4 C++ classes*

The netCDF-3 hierarchy, though not adopted in its entirety, was used to facilitate a smooth transition from netCDF-3 to netCDF-4. NcFile representing a File, NcDim representing a Dimension, NcAtt representing an Attribute, NcVar representing a Variable and NcValues, the accessing class for blocks of unknown types of data, were adopted to facilitate an easy transition from the netCDF-3 C++ interface to the netCDF-4 C++ interface. The NcGroup class was added to enable the creation and manipulation of group objects, and the NcException class was created to throw exceptions when an error is returned from the C interface, to facilitate efficient error handling.

Unlike the NcValues class in netCDF-3, netCDF-4’s NcValues class also acts as a combination of the netCDF-3 C++ API’s NcValues class and Fei Liu’s wrapper library class. Information is still stored in a C style array; however numerous overloaded operators have been added to the interface in order to facilitate efficient data access.

During the project, questions of how to represent user-defined variables arose. We considered two different options. The first was to modify the NcVar class to handle variables of both primitive and user-defined types, and the second was to create a class to handle primitive
type variables and a separate class to handle each kind of user defined variable. After careful consideration we opted to implement the first option for simplicity - to avoid the creation of a large number of software components needed to implement the second option - and also to remain faithful to Arnold’s philosophy that the interface should be intuitive. The first option allows users to see user defined type variables simply as an extension of primitive type variables.

To reduce code duplication that would be present if we allowed the NcVar class to treat atomic/primitive type variables differently from user defined type variables, we created a base class NcType to handle all type needs. We then created an NcAtomicType class to handle primitive types and the NcUserDefinedType class, both of which inherit from the NcType class. The NcUserDefinedType acts as the base class for all other user defined types, NcCompoundType represents compound user defined types, NcEnumType represents enumerated user defined types, NcOpaqueType represents opaque user defined types, and a NcVLenType classes handles the creation of variable length user defined types. The NcVar class simply takes an NcType object pointer when creating a variable.

When necessary, functions in each of the classes mentioned above were templated to make the interface more generic. As requested via the netCDF mailing group, numerous methods in each of the classes were made virtual, and all the classes, variables, and constants used were placed in a netCDF namespace to decrease collisions with other namespaces. These aspects of the netCDF-4 interface design should ensure that it serves as a good interface for C++ programmers while existing as a thin layer above the C interface. We also determined, based on user feedback, that it was essential to provide iterators for groups, dimensions, attributes and variables. Thus, a nested iterator class was included in the NcGroup, NcDim, NcAtt, and NcVar classes.

To represent variable length user defined variables and ragged arrays, we decided simply to store the minimal amount of information necessary in each class and rely on the C representation of such variables. The information required includes a pointer to allocated space for values and a length specifying how many values are in the variable-length data. A function is proved to later free the allocated space.

After completing the initial design we conducted a design review and concluded that netCDF variables, groups, dimensions, and attributes should not exist independently of a netCDF file and thus made all constructors except the file constructor and the user defined type constructors private.

While refining the design, we established three criteria for accepting any change to the interface design: the change had to make the interface more efficient, it had to increase the intuitiveness of interacting with the interface, and implementing the new modification had to take no more than the time allotted for the specific section of the project. Throughout the project, we decided to use test driven development because it has been proven to be the most effective testing scheme in software development. Test driven development incorporates testing into the development cycle [6], and is also how we incorporated Arnold’s philosophy of testing into our development process.
Two major questions surrounded the design of the interface. The first was how much direct access a user would be allowed to different portions of the interface. The second was how to reduce redundancy in code by using templates without using type casting inside the methods before calling down to the C interface. We opted to limit the users’ access to the NcGroup, NcDim, NcAtt, and NcVar classes to limit the error checking responsibilities of the interface. We also decided to reduce the redundancy in the previous C++ interface which used macros by using template functions. Before moving on to full implementation we made sure that our stub methods and header files compiled without errors, and we also did preliminary implementations of the most basic methods.

3.2 Implementation Methodology

While implementing the C++ application programming interface we decided to create nested classes within the NcGroup class and a nested class within the NcVar class to serve as iterators for attributes, variables, groups, and dimensions. Each class (NcGroup, NcVar, etc.) keeps track of each class instance created within a standard template library map. The group class keeps track of each dimension, attribute, variable, and subgroup added to the group in standard template library maps and uses map iterators to traverse over objects that have been created. We also decided to have the variable class keep track of each attribute added to the class and traverse over it using the standard template library map iterator.

While implementing the C++ application programming interface and determining the levels of dependency among all the classes, we discovered that allowing groups to exist separately from files would place a heavy error checking burden on the NcGroup class. Instead of having the NcGroup class check to see if a valid file was passed as an attribute when requesting a new group, we opted to create groups only from the file class. After successfully implementing this portion of the interface we also used a similar process when dealing with attributes, dimension, and variables that can be added to a group. An attribute can only be created by the NcGroup or NcVar class. To accomplish this, we denied public access to the NcGroup constructor and made the NcFile class a friend of the NcGroup class to allow the NcFile class to create new groups. Other constructors were implemented in a similar way.

Section 4. Results and Discussion

The interface we have designed and implemented meets the requirements of a lightweight C++ interface for the netCDF-4 data model and data format. Modifications to the original design made during implementation include creating constructors to handle reading file information, and a technique which makes retrieval, storage, and data traversal very efficient was the use of the STL map class. At present the netCDF-4 C++ application programming interface is not backward compatible due to the time constraints of the project. In particular, it is not able to read netCDF-3 and older netCDF file formats. However, the interface has been implemented and is available for download at http://www.unidata.ucar.edu/software/netcdf/netcdf-4/. We are currently working on additional test cases. The interface currently passes all tests.
We have performed a small scale test to ensure that all aspects of the API were functional. In addition the C++ interface, unlike the C interface, does not feature parallel I/O given its complexity and time constraints of the project.

A variety of netCDF-4 files were created to show that groups, variables, dimensions, and attributes were successfully created. Demonstrating that the implemented netCDF-4 C++ interface is robust was an important aspect of this project. We created a small test suite to evaluate the NcFile, NcGroup, NcVar, NcDim, and NcAtt classes and to evaluate the robustness of the netCDF-4 C++ interface. Though small, the test suite was able to demonstrate the correctness of the interface and promote Unidata’s test based development approach. In addition to the test suite, we also used the interface to create a set of examples normally distributed with each new netCDF release to show users how to store the same information in netCDF format using different interfaces. The netCDF-4 C++ interface will allow data providers and developers, with a preference for C++, to take advantage of the new features netCDF-4 offers for creating portable, self-describing datasets.

Section 5. Conclusions and Future Modifications

Overall the project has demonstrated a simple procedure for creating a middleware library that is useful, updatable, and that makes use of the newest standards available. Areas for further improvement include considering other options for implementing iterators, further consideration of which methods should be public or private, and also deprecating options that were implemented for backward compatibility to streamline the code.

Future modifications of this project include releasing the netCDF-4 C++ API to the netCDF community for larger testing and its incorporation into commercial software. It is also important to conduct a survey of early users of the API to get feedback about the design. The survey could help determine, for example, if the name collision conflicts they experienced with portions of the netCDF-3 C++ interface were resolved, if backward compatibility is really necessary in the new interface, or if a netCDF-3 API should exist separately from the netCDF-4 API.

Possible modifications to the C++ API include adding parallel write capabilities, and also adapting aspects of the interface for an embedded netCDF-4 API. In addition to the aforementioned modification another layer could be built on top of the netCDF-4 C++ API to serve as a C++ API for libCF, a library designed to standardize conventions used in netCDF files to facilitate wide data reuse within the atmospheric and geoscience community.

Acknowledgements:

This work was funded by the SOARS Program. We would also like to acknowledge the efforts of Russ Rew and Anne Wilson at the Unidata Program Center –UCAR in Boulder, Colorado.
References:


