Limit Point Systems, Inc.

# Introduction

This document shows how to use key features of the SheafSystem. The C++ examples in the document are available as source code in the examples subdirectory of the SheafSystemProgrammersGuide module and the reader is encouraged to build and execute the examples along with reading the text. The examples are numbered and the source for example N is in exampleN.cc. There are a few compilation and execution examples in the text, these are given in Linux using the csh shell, Gnu C++, and Gnu make.

# What you'll need

To take full advantage of this document, you'll need a few things in addition to the document itself, namely:

* an installed copy of the SheafSystemProgrammersGuide module, which includes the examples,
* a C++ compiler,
* a web browser, and
* an installed copy of the SheafSystem.

# The SheafSystem installation

The SheafSystem installer installs all the files of the SheafSystem in a directory tree. We will have to refer to the root of this directory tree repeatedly, so to simplify the notation, we'll let <sheaf\_dir> refer to the full path to the root directory of the installation, for instance:

<sheaf\_dir> = /usr/local/SheafSystem

Wherever you see <sheaf\_dir> in this document, mentally replace it with the full path to your SheafSystem installation.

The installation includes 4 configurations of the libraries: Debug-contracts, Debug-no-contracts, Release-contracts, and Release-no-contracts. The "Debug" configurations are unoptimized and contain symbol information for use by interactive debuggers such as gdb. The "Release" configurations are optimized and contain no debugging information. We'll describe "Contracts" below. Generally speaking, the Release configurations are higher performance that the Debug configurations and the no-contracts configurations are much faster than the contract configurations.

The examples will compile and execute with any configuration, but we will always use the Debug-contracts configuration in the text below.

# Getting started

## PartSpace metaphor

The PartSpace document describes the fundamental concepts of the SheafSystem in non-mathematical terms using the common notion of basic and composite parts, tables, and table schema. This document assumes the reader is familiar with the PartSpace metaphor.

## Sheaf tables

As described in the PartSpace document, a SheafSystem database is a collection of tables. Each table is equipped with a covering relation graph describing the lattice order of its rows and another graph describing the lattice order of its columns. Each such object table has an associated table called its schema table and the row graph of the schema table defines the column graph of the object table. A member of the row lattice is represented by a node in the row graph. A member also has a corresponding row in the table if and only if it is a basic part, a join irreducible member (JIM) in the row lattice.

There are 3 special tables. the primitive schema table, the primitives table, and the namespace table. The primitives schema table terminates the schema recursion, it is its own schema table. The primitives table describes each primitive type supported by the system.

## Namespaces

A namespace table is a special table in each database that serves as a container and table of contents for all the other tables. The SheafSystem includes 3 predefined namespace types: the sheaves\_namespace, the fiber\_bundles\_namespace, and the geometry\_namespace. Each of these predefines the sheaf schema for the C++ types defined in the sheaf, fiber\_bundle, and geometry components, respectively. (The fields component doesn't have its own schema).

Creating an instance of a namespace is typically the first thing a client must do to use the SheafSystem, so we start with an example of how to do that using the most basic namespace, sheaves\_namespace. This example will also cover the basic mechanics of compiling and linking with the SheafSystem.

### Example 1: Hello, Sheaf

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

// Create a standard sheaves namespace.

sheaves\_namespace lns("Hello-sheaf");

// Write its name to cout.

cout << lns.name();

return 0;

}

This code is in the SheafSystemProgrammersGuide module in examples/sheaf/example1.cc along with a Makefile:

#

# Full path to your C++ compiler, for instance /usr/bin/g++

#

CXX = /usr/bin/g++

#

# Full path the SheafSystem installation include and library directories

#

SHEAF\_INC\_DIR =<sheaf\_dir>/include

SHEAF\_LIB\_DIR = <sheaf\_dir>/Debug-contracts/lib

example1: example1.cc

$(CXX) -o example1 -I$(SHEAF\_INC\_DIR) -L$(SHEAF\_LIB\_DIR) example1.cc -lsheaves

To compile and link the example, you first have to configure the Makefile to your installation by setting the 3 variables CXX, SHEAF\_INC\_DIR, and SHEAF\_LIB\_DIR to the actual values for your installation. Then we can compile and link by:

>make example1

This command will compile example1.cc and link it with the shared library libsheaves.so to create an executable example1 in the current directory. We have to tell the dynamic loader where to find the shared library by setting the environment variable LD\_LIBRARY\_PATH to contain the path to the SheafSystem library directory, that is, the same value we set SHEAF\_LIB\_DIR to in the Makefile, for instance:

>setenv LD\_LIBRARY\_PATH <sheaf\_dir>/Debug-contracts/lib

Now we can execute the example:

>./example1

Hello-sheaf

That's the basic mechanics of creating an application with the SheafSystem. We've created a sheaves\_namespace in this example, but before we can do much with it, we need to learn a few programming patterns that the SheafSystem uses repeatedly.

# Programming patterns

There are a few design features shared by all the classes in the SheafSystem. In this section we will give a quick introduction to the most ubiquitous of these patterns. We'll introduce some more patterns later, as we need them, and also go into some of these initial patterns in more detail.

## Design by contract

The sheaf system is implemented using the "design by contract" programming paradigm. We'll cover the essentials of the method and how they are used in the SheafSystem. For a more detailed introduction, see the excellent book Design By Contract, by Example by Richard Mitchell and Jim McKim.

When using design by contract, each class is equipped with an invariant, a set of assertions that must be true at any time control returns the client. (The invariant is not defined when control is within a member function of the class.) Every member function is equipped with preconditions and postconditions. The preconditions are assertions that must be true when control enters the member function; the postconditions must be true when control leaves the member function. The "contract" in "design by contract" is between the client and the member function: if the client guarantees the preconditions are true, the member function ensures the invariant and the postconditions are true.

The invariant, precondition, and postcondition assertions are specified using "invariance", "require", and "ensure" macros, respectively, in the source code. If contracts are enabled when the library is compiled, these clauses will be evaluated as part of the execution of the member functions. If the conditions specified in the clauses are not true, execution throws an exception with an error message, which usually terminates the program.

The contracts are extremely useful for detecting improper use of the classes and member functions and are thus an important debugging tool. Once client code is correct, the contracts can be disabled to improve efficiency.

The SheafSystem Debug-contracts and release-contracts configurations are compiled with contracts enabled. The Debug-no-contracts and Release-no-contracts are compiled with contracts disabled.

The contracts are also published as an essential part of the reference documentation and are critical to using the sheaf system correctly. Let's look at the reference documentation for the sheaves\_namespace constructor we used in example1. The reference documentation is generated in html, so you can open it with your browser. The main page is <sheaf\_dir>/documentation/C++/index.html. If you browse to the documentation for class sheaves\_namespace and click on the constructor sheaves\_namespace(const string& xname), you'll find:

sheaf::sheaves\_namespace::sheaves\_namespace ( const string & *xname* )

Creates a sheaves namespace with name xname.

Precondition

* poset\_path::is\_valid\_name(xname)

Postcondition

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)
* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname
* !in\_jim\_edit\_mode()
* [host()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#adc8f6d6d2b952a6842a1d09de75bff9a) == 0
* !index().[is\_valid()](http://192.168.4.199/comp-tutorial-dev-4/df/d4b/namespacesheaf.html#a3dd8f96a360e1b63c6caa744e5ccf7b3)
* [index()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#a9a283b1819bc8e75b212bff26fc645b0).same\_scope(member\_hub\_id\_space(false))
* [has\_standard\_subposet\_ct()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#af5786ce90013ec6e72dbacd9b67e1c13)
* [current\_namespace()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#ae8ca3a11bc745cf0b275a70ab71b2d70) == this
* [state\_is\_not\_read\_accessible()](http://192.168.4.199/comp-tutorial-dev-4/d1/d3c/classsheaf_1_1read__write__monitor__handle.html#adc32a6090b2df1e5673444d5170539f3)

So what does this tell us? The precondition:

* poset\_path::is\_valid\_name(xname)

tells us exactly what conditions the argument xname has to satisfy if we want this call to the constructor to work correctly, namely is\_valid\_name(xname) has to be true. Well, what does that take? If we look up poset\_path∷is\_valid\_name we find:

static bool sheaf::poset\_path::is\_valid\_name( const string &  xname )

True if xname is not empty and contains only name legal characters.

Postcondition

* result == (!xname.[empty()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a05ee8f14bcc22701b551059341f16749) && (xname.find\_first\_not\_of([name\_legal\_characters()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a34019af3a5bee6f34d3ec2c2657a8671)) == string::npos))

So xname can't be empty and can't contain any characters not in name\_legal\_characters(). If we click on name\_legal\_characters we find:

static const string & sheaf::poset\_path::name\_legal\_characters( )

The characters a name is allowed to contain.

Postcondition

* result == "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_ -,.=+()\*:?"

So xname has to be non-empty and contain only the above characters.

If xname satisfies these conditions, which it does in example1, then the postcondition gives a great deal of information about what state the sheaves\_namespace object is in after construction.

The first postcondition is:

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)

that is, the invariant has to be satisfied. As we said above, this is an implicit postcondition of every member function, even if we don't explicitly provide it as part of the contract. So what does this mean for sheaves\_namespace? Well, click on invariant() to find:

virtual bool sheaf::namespace\_poset::invariant ( ) const

Class invariant.

Invariant

* poset\_state\_handle::invariant()
* host() == 0
* !index().is\_valid()
* !is\_external()
* is\_attached() ? primitives().is\_attached() : true
* is\_attached() ? (primitives().index() == PRIMITIVES\_INDEX) : true
* state\_is\_read\_accessible() ? primitives().state\_is\_read\_accessible() : true
* is\_attached() ? primitives\_schema().is\_attached() : true
* is\_attached() ? (primitives\_schema().index() == PRIMITIVES\_SCHEMA\_INDEX) : true
* state\_is\_read\_accessible() ? primitives\_schema().state\_is\_read\_accessible() : true

Sheaves\_namespace inherits namespace\_poset and doesn't override the invariant, which is a virtual function, so the invariant of sheaves\_namespace is the invariant of namespace\_poset. The invariant in a derived class must be at least as strong as the invariant in the base space, so the invariant of namespace\_poset calls the invariant of its base class, poset\_state\_handle. Beyond whatever poset\_state\_handle∷invariant() ensures, the namespace\_poset invariant ensures several properties of the data members, primitives() and primitives\_schema() in particular.

As this invariant shows, the conditional expression

* x ? y : true

appears frequently in the contracts, so it is worth describing in more detail. As an assertion, this expression can be read "x implies y", that is, x can be either true or false, but if is true, then y must be true as well. If x is false, there is no condition on y.

The reader is encouraged to examine the poset\_state\_handle invariant to learn what additional invariances sheaves\_namespace has inherited, but we'll move on to the rest of the postcondition of the constructor. The next postcondition is one you'd expect:

* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname

that is, the name of the namespace is the name we gave it.

The remainder of the postconditions ensure various arcane properties of the namespace that we're not very interested in right now. But when your tackling a tough debugging problem, any of these may be just the piece of information you need!

The power of the design by contract method comes from the great amount of detailed information contained in the assertions and two further properties. First, if contracts are turned on, that is if you are using either the Debug-contracts or Release-contracts configuration of the library, the pre- and post-conditions of a function are executed whenever the function is called. Second the contracts exhibited in the documentation are extracted directly from the source code. The combination of the two allows you to reason about the behavior of the code with great confidence while designing, programming, and especially while debugging.

So what happens if the contract for a member function is not satisfied? Let's find out by trying to create a sheaves\_namespace without a name.

### Example 2: contract for sheaves\_namespace constructor.

#include "sheaves\_namespace.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example2:" << endl;

// Attempt to create a standard sheaves namespace

// with an empty name. This violates the preconditions

// of the constructor and will throw an exception and abort.

sheaves\_namespace lns("");

// Done.

return 0;

}

If we compile and run this, assuming we still have LD\_LIBRARY\_PATH set from running example1, we get:

>make example2

>./example2

terminate called after throwing an instance of 'std::logic\_error'

what(): 'poset\_path::is\_valid\_name(xname)' in file namespace\_poset.cc at line 1941

Abort

The error message tells you exactly what assertion failed. If you're debugging, you can walk back up the stack from where the exception was actually thrown to the assertion that failed and inspect local variables, for instance xname, to determine what went wrong.

## Concurrency control

One of the attractive features of the sheaf data model is that its mathematical formalism provides a natural language for describing concurrency and parallelism. The sheaf system libraries were designed for concurrent programming using an access control mechanism based on the monitor design pattern. Currently, this mechanism is only partially implemented and the SheafSystem libraries are delivered with the access control mechanism disabled. Programmers nevertheless must be aware of certain aspects of the access control mechanism, which we describe in this section. More complete examples are included in Appendix A.

Access to every table is controlled. A client thread can have no access, read access, or read-write access. At any given time, either no client has access, exactly one client has read-write access, or one or more clients have read access. Before reading or writing a table or any of its members, a client must request read access or read-write access, respectively. After accessing the table, the client must release access. If a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access.

The concurrency control mechanism is "enforced" through precondition clauses in the table member functions. In order to make concurrency control apparent to the client and avoid dead lock, the library routines do not themselves request or release access without the client knowing it. Instead, they "publish" their access requirements as preconditions and let the client control the access.

For instance, in example 1 we invoked sheaves\_namespace∷name(). Consulting the reference documentation, we find for the name() member function:

virtual string sheaf::namespace\_poset::name() const

The name of this namespace.

Precondition

* state\_is\_read\_accessible()

So, if the access control mechanism is enabled, the client must request read access, invoke the name function, and release access:

lns.get\_read\_access();

cout << lns.name() << endl;

lns.release\_access();

Getting and releasing access can be a tedious programming chore. Furthermore, it is syntactically impossible in some cases, for instance within a pre- or post-condition clause. So many member functions offer an "auto-access" option. These routines will automatically get and release the access they need, if the client allows it by setting an auto-access argument to true. If invoked with the auto-access argument false, the client must get the required access before making the call. These routines also publish their access requirements as preconditions. For instance, the auto-access version of the name function is:

virtual string sheaf::namespace\_poset::name( bool  xauto\_access ) const

The name of this namespace.

Precondition

* state\_is\_auto\_read\_accessible(xauto\_access)

Using this version of the name function, the client need only invoke the function with argument "true":

cout << lns->name(true) << endl;

The function will request read access, get the name, release access, and return the name.

When the access control mechanism is disabled, the client always has read-write access and neither requesting nor releasing access is necessary. Functions with an auto-access argument can be called with either true or false, either will work. However, the access control mechanism doesn't quite disappear from the programmer's view. The auto-access signatures are still present and the access requirements still appear as preconditions in the contracts.

## Handles and states

The Sheaf System is object-oriented, so the client interacts with the library by manipulating the various objects presented by the library interface. Lattice members are a prime example. Many of the objects exported by the interface are not however stored as explict objects internally. Both memory and performance efficiency often require that such objects be implicit - stored as disjoint data items in bulk arrays. The problem of how to present an externally explicit object interface to an internally implict object is a common software design problem and several similar design patterns - flyweight, proxy, surrogate, etc, have been developed to address this problem. In the Sheaf System, we call such a surrogate object a handle and the internal data it accesses is called its state.

For the most part, the distinction between handles and states is an implementation detail that the client needs to be only vaguely aware of. The client uses the handle object as if it were stored internally without worrying about the internal details. But there is one aspect the client has to be aware of: the client has to somehow get a handle to the desired object and when finished with it the client may have to explicitly release it.

There are two basic patterns. In the first pattern, some explicit object has a data member which is a handle and it provides an accessor to this data member. For instance, sheaves\_namespace, like every lattice, has a top member. This member is represented by a data member which is a handle and sheaves\_namespace exports an accessor:

sheaves\_namespace lns;

namespace\_poset\_member& ltop = lns.top()

The namespace object allocated and owns the handle. The client need not and should not worry about releasing or otherwise deallocating the handle.

The second pattern addresses the more general case in which the number of handles the client needs and what states they should be attached to is not known at compile time. In order to support efficient allocation and deallocation of handles, the system maintains pools of handles which the client can "borrow", use, and return. For instance, we'll see in the next section that index spaces are accessed via handles and the client can get a handle from the appropriate index space family:

index\_space\_handle& lids =

lns.member\_id\_spaces(true).get\_handle("member\_poset\_id\_space");

When accessed in this way, the handle must be released when the client is finished with it:

lns.member\_id\_spaces(true).release\_handle(lids, true);

How does a client know whether to release a handle or not? Simple, if you got the handle by calling get\_<whatever>, you need to release it by calling release\_<whatever>. Release if and only if get!



Figure 1: Hub and spoke architecture of an index space family.

## Index spaces and scoped indices, part 1

The members of the row lattice of a table (and hence the members in the column lattice as well) are identified by integer ids. Subsets of the members are very important in the SheafSystem and it is frequently useful to generate a special purpose index scheme for a given subset. Such an index scheme is referred to as an "index space", or "id space" for short. The SheafSystem provides extensive support for defining and using id spaces.

### Index spaces and iterators.

More specifically, an index space is a set of integer ids. The system supports the creation and use of a family of index spaces. The fundamental id space of the family is the member id space - the ids automatically generated for the nodes in the row graph. This index space is called the hub id space because the index space family has a hub and spoke architecture as shown in Figure 1. As you can see from the diagram, there are several different kinds of id space and even two hub id spaces, the "unglued" and "glued" versions. We'll describe this structure and how to create and modify id spaces later. For the moment, hub id space means unglued hub id space and you should just think of each id space on the rim as a way of indexing some subset of the hub id space, with each spoke representing a map. We'll focus on the basics of how to use the id spaces automatically created by the system.

As one might expect, the principal use for a member id is to access the features of the member the id refers to. The principal use of a member id space is to iterate over all the members in the subset defined by the id space. Let's look at an example.

### Example 3: Iterates over the member hub id space.

#include "hub\_index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example3:" << endl;

// Create a standard sheaves namespace.

sheaves\_namespace lns("Example3");

// Get a handle for the member hub id space.

const index\_space\_handle& lmbr\_ids = lns.member\_hub\_id\_space(true);

// Find out how many ids are in the id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

// Id spaces are defined as half open intervals, like STL iterators.

// If the space is "gathered", begin() == 0 and end() = ct().

// If the space is not gathered, it's "scattered".

cout << "begining at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

// The main thing one does with id spaces is iterate over them.

// Get an iterator from the iterator pool.

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// The current member of the iteration is "pod()".

// "POD" is an ISO C++ acronym for "plain old data".

// A pod is an ordinary integer id, in contrast with

// a "scoped\_index" id, to be discussed shortly.

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id, but since we're using

// the hub id space, pod and hub pod are the same thing.

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// You can reuse an iterator by resetting it.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// If you got an id space or iterator from the pool with get\_

// you have to return it to the pool with release\_.

lmbr\_ids.release\_iterator(lmbr\_itr);

// The id space itself is a data member of the id space family,

// we didn't get it from the pool with get\_, so we don't have to

// release it.

// Exit:

return 0;

}

If we execute example3 we get:

>./example3

SheafSystemProgrammersGuide Example3:

\_\_hub has 6 ids.

begining at 0 and ending at 6 gathered

Iterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

Reiterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

### Id maps and scoped ids.

As we said above, id spaces are used for indexing subsets. For instance, in a namespace, the member poset id space indexes just the jims, which represent the member posets - the other posets contained in the namespace. There may be several or even many id spaces available in a practical setting. Various member functions may require an index to be in a particular id space, most commonly in the hub id space. The id maps associated with the spokes in the id space family provide the mechanism for translating between id spaces.

Every id space has a map to the (unglued) hub id space. The index\_space\_handle class provides member functions for mapping ids between the id space and the hub id space:

pod\_type hub\_pod (pod\_type xid) const

The pod index in the unglued hub id space equivalent to xid in this id space; synonym for unglued\_hub\_pod(pod\_type).

and

pod\_type pod (pod\_type xid) const

The pod index in this space equivalent to xid in the hub id space.

Using these functions we can map between id spaces. For instance, if id1 is an id in id\_space1 and id\_space2 is different id space, then

pod\_type id2\_eqv\_1 = id\_space2.pod(id\_space1.hub\_pod(id1));

is the id in id\_space2 that identifies the same member identified by id1 in id\_space1, if such an equivalent member exists. the postcondition for the pod(pod\_type) function is:

* !is\_valid(result) || contains(result)

So if id\_space2 does not have an equivalent member, id2\_eqv\_1 is assigned an invalid value.

Using these functions, the programmer can map ids between id spaces. But it can be tedious. Worse, it may be difficult or impossible for a programmer to track just what id space a given index is in. The scoped\_index class provides a convenient mechanism for both managing the connection between an id and the space it belongs to and for automatically mapping between id spaces. We call the id space an id belongs to the scope of the id. A scoped\_index is a a pair (id, scope). Most member functions that require an id as input are available in two signatures; one signature that takes a pod id and one that takes a scoped id. One can use a scoped id, once it has been initialized, without worrying what scope it is in; any function that accepts a scoped id will translate it to the scope it requires. We'll see more complex examples of mappoing between id spaces later, for now let's redo example3 using the member poset id space, the hub id space, id maps and scoped ids.

### Example 4: Iterates over the member poset id space.

#include "index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example4:" << endl;

sheaves\_namespace lns("Example4");

// Get a handle for the member poset id space;

// has one member for each poset in the namespace.

const index\_space\_handle& lmbr\_ids =

lns.get\_member\_poset\_id\_space(true);

// Print out the same info we did for the hub id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

cout << "begining at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id which we can get in two ways.

// First, using the map from the id space to the hub id space.

index\_space\_iterator::pod\_type lhub\_pod = lmbr\_ids.hub\_pod(lpod);

// Second, the iterator can provide the hub id equivalent for

// the current id and it can be faster because for some id space

// types it can avoid the map lookup.

lhub\_pod = lmbr\_itr.hub\_pod();

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lmbr\_itr.hub\_pod(), true);

cout << (lns.is\_jim(lhub\_pod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// If you don't want to think about what the scope for an argument

// should be, you can use the scoped\_index signature.

// Reset the iterator and re-iterate using

// the scoped\_index signature for member\_name.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// Create a scoped id for the current member of the iteration.

scoped\_index lscoped\_id(lmbr\_ids, lmbr\_itr.pod());

// Use the scoped\_index signature id to get the member name.

cout << "scoped\_id: " << lscoped\_id;

cout << " name: " << lns.member\_name(lscoped\_id, true);

cout << (lns.is\_jim(lscoped\_id) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

lmbr\_ids.release\_iterator(lmbr\_itr);

// Exit:

return 0;

}

When we run example4 we get:

>./example4

SheafSystemProgrammersGuide Example4:

member\_poset\_id\_space has 3 ids.

begining at 0 and ending at 3 gathered

Iterate:

id: 0 hub id: 2 name: primitives\_schema is a jim.

id: 1 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 2 hub id: 4 name: primitives is a jim.

Reiterate:

scoped\_id: (2, 0) name: primitives\_schema is a jim.

scoped\_id: (2, 1) name: namespace\_poset\_schema is a jim.

scoped\_id: (2, 2) name: primitives is a jim.

# Storage\_agent

We've talked about the notion of a SheafSystem database, so there must be some way to make a namespace persistent and indeed there is. Persistent storage is managed by the storage\_agent class. A storage\_agent makes it particularly easy to save an entire namespace to disk, as we show in the next example.

## Example 5: Write a namespace to a file

# Viewing Namespaces

## Stream insertion operator

### Example 6: Write namespace to cout

Write name to std out as a poset and as a namespace.

## Read.t

### Example 7: View namespace with read.t

## SheafScope

### Example 8: View namespace with SheafScope

# Posets

## Example 9: Creating, accessing, and deleting posets

## Creating posets

Create a poset, write to cout.

## Accessing posets

Three common signatures, poset path

### Example 10: Access poset and write to cout.

## Deleteing posets

Delete the poset.

# Poset members

## Example 11: Creating andd manipulating poset members with the poset interface

## Creating poset members

Create a jim

## Accessing poset members

Naming, various queries

## Ordering poset members

## Deleting poset members

## Handles

Repeat all the same examples with handles

## Schema posets

# Concurrency control examples

The access control mechanism is a work in progress. The control mechanism itself is complete and is implemented both for multiple threads using pthreads and for single threads. When the library is compiled with threads enabled and a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access. When the library is compiled with threads disabled, requests do not block, they return immediately. The library is currently delivered with threads disabled because the use of threads and concurrency in the library is only partially implemented and not tested. The access control mechanism is disabled by default but can be enabled by the programmer. These examples demonstrate use of the manual and auto-access mechanisms.

## Example A1: manual access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA1:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA1");

// Write its name to cout.

// Requires read access to the namespace.

// Be polite, request access.

// If threads are enabled and another thread has

// read-write access, execution will block until it

// releases access. Otherwise, the request will succeed

// immediately.

// You can nest requests as deep as you want, or at least

// until the integer depth counter overflows.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Invoke the operation.

cout << lns->name() << endl;

// Be proper, release access so this thread

// or another can get write access.

// Have to match every request with a release.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Delete the namespace, requires read-write access.

// Be polite, request access. If threads are enabled

// and another thread has either read or read-write

// access, execution will block until it releases access.

// Otherwise, the request will succeed immediately.

// This client must not already have read-only access,

// see precondition for details.

lns->get\_read\_write\_access(false);

// Invoke the operation.

delete lns;

// Deletion is the only case where the client

// can not be proper and release access.

// Create another namespace.

lns = new sheaves\_namespace("Example3B");

// Invoking a function that requires access

// without first getting access violates the

// precondition of the function.

// The following will throw an exception and abort.

cout << lns->name() << endl;

return 0;

}

If you compile and run example A1, the output is:

SheafSystemProgrammersGuide ExampleA1:

request depth 0

request depth 1

request depth 2

Example3A

request depth 2

request depth 1

request depth 0

terminate called after throwing an instance of 'std::logic\_error'

what(): 'is\_external() ? name\_space()->state\_is\_read\_accessible() : state\_is\_read\_accessible()' in file poset\_state\_handle.cc at line 1178

Abort

## Example A2: auto-access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA2:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA2");

// Write its name to cout.

// Requires read access to the namespace.

// Invoke the auto-access version of the operation with

// auto-access set to true.

// Operation will request and release access as needed.

cout << lns->name(true) << endl;

return 0;

}

If you compile and run example A2, the output is:

SheafSystemProgrammersGuide ExampleA2:

ExampleA2