ObjectStore and STL

By Adrian Marriott & Ben Rousseau

Introduction

The main aim of this paper is to show how easily STL can be used with ObjectStore. We describe how to store STL containers, their iterators and STL basic_strings persistently within ObjectStore. A standard STL allocator in *principle* should work with any STL implementation, but in practice this will depend on how standards-compliant any particular STL implementation is. We present a fully working ObjectStore allocator designed for the popular STLport freeware implementation that has been tested to capacities of three million objects or more; storing objects, running STL algorithms such as sort and reverse, and searches using find_if methods. We describe factors that influence the design and implementation of the allocator, show how simple it is to use and then describe some of the potential pitfalls that may occur when storing STL persistently and how to solve them.

Intended Audience

Technical programmers who are using or intending to use STL with ObjectStore. It is assumed that the reader is familiar with ObjectStore terms such as os_cluster and os_typespec. For more information on these subjects and ObjectStore in general see [1] below.

Requirements

The minimum requirements to use standard off-the-shelf STL implementation within ObjectStore are:

1. Use ObjectStore v6.1 or above
2. Use an STL implementation that is 'allocator compliant'. This means that internally, it will *always* use the allocator interface for all memory management and never call 'new' or 'delete' directly when creating or destroying its own internal data structures.
3. Compiler must provide RTTI support in such a way that the class size is not changed. Compilers commonly provide a compile-time option which can enable or disable RTTI support and we require that this option leave the class size unaltered. This is academic for most of the ObjectStore supported platforms, since most encode RTTI information in the virtual pointer or some other way.

These three points are necessary but not sufficient. There may be additional 'interface code' that is required to overcome the limits of some compiler's template handling capabilities.

STL Allocator for ObjectStore

The allocator presented here is designed for the popular STLport implementation and is based on a typical user-defined allocator [2]. Only a single method is changed compared to a 'default' allocator example: the method `MyAllocator::allocate()` must be defined so that the memory returned is within ObjectStore's Persistent Storage Region (PSR). For clarity it is necessary to distinguish here two categories of object before we proceed:

1. **STL Internal Objects**. These are objects internal to an STL collection at the implementation level, such as the list nodes or C++ arrays used behind the scenes. These objects are usually invisible to the application programmer and comprise the STL itself.
2. **User Objects**. These are objects which are defined by the user at the application level and can be inserted into STL containers. For example, an object of type Foo could be used within an STL vector of type `vector<Foo>`. The user objects are the Foo objects within the vector.

Any allocator will need to return memory suitable for both these categories of objects. ObjectStore has various overloadings of the placement new operator to allocate persistent memory, and this means when writing the `allocate()` method there are two problems to surmount:

1. We need an **os_cluster** object.
2. We need the correct **os_typespec** object.

We examine these in turn now.

Finding the Correct ObjectStore Cluster

An ObjectStore cluster is required into which we can allocate our memory. Without this, the memory cannot be located within the database. The simplest strategy is to assume that all STL internal objects can usefully be located within the same cluster, and that this cluster should be the same one that holds the STL container object itself. This is very easily achieved with a call to the static method `os_cluster::of(this)` inside our allocator.

What is the justification for this simple approach? Well, it is safe to assume that any use-case that accesses the STL container, either for read or for write, is going to fetch and lock the page holding the container itself. It is also sound to assume that, in order to use the container in any way, at least a sub-set of the STL internal objects managed by the container will be accessed, so locating these objects within the same cluster as the container will almost always make sense, (although there may be exceptions and some of these are addressed below). So for STL internal objects this approach is reasonable.

Using `os_cluster::of(this)` alone will also result in the user objects being located in the same cluster as the container itself, so the pages of the cluster will contain STL internal objects interspersed with user objects. This might present less than optimal locality of reference for certain critical use-cases. The simple solution is to store pointers to the user objects in the STL collection, not the objects themselves, and locate the user objects in remote clusters as required. So create `vector<Foo*>` rather than `vector<Foo>`. The extra pointer dereference in performance terms will in most cases be 'lost in the noise' and these pointers-to-Foo, although actually 'user objects' can be treated for clustering purposes as 'internal'; it makes sense that these are allocated with the STL internal objects. By storing pointers, or in some cases soft-pointers, in the STL collections and locating the objects themselves away from the collection we can separate out the clustering issues in a simple and direct way outside the allocator.

Another positive aspect of using `os_cluster::of(this)` is speed. There is no faster way to get hold of a suitable cluster.

Finding the Correct ObjectStore Typespec Object

It is necessary to provide the correct ObjectStore typespec object which corresponds to the class being allocated and this must be achieved this in such a way that typedefs, classes, structs, templated types (some with multiple arguments), native types and pointers to any of these can all be accommodated. Additionally it must not be assumed that a `get_os_typespec()` method exists for these classes, because the authors of general purpose STL implementations will not have declared these. The approach taken here solves all these problems and introduces near-zero runtime overhead.

The correct ObjectStore typespec object is retrieved using a new `ts<T>()` function template the details of which are shown in the appendix. Each typespec object is managed by a dedicated transient singleton [3], implemented as a template. The algorithm takes advantage of the os_typespec constructor that can be passed a 'stringified' representation of the class name and proceeds as so:

1. Check static os_typespec pointer data member to determine if the requested typespec object has already been created. If so return it directly.
2. If not use RTTI and glean the name of the class from the template argument T passed into `ts<T>()` call.

3. Clean this name so that it maps to the corresponding name required by the os_typespec constructor. This mapping is systematic and relatively straight-forward.
4. Create the typespec object on the heap and target the static os_typespec pointer data member thereto.
5. Return the os_typespec pointer

Now it is clear where the efficiency comes from, as each typespec object is only created once in the lifetime of a process. This also explains why there is a dependency on compiler support for RTTI.

ObjectStore STL Allocator Implementation

Now we can access a suitable ObjectStore cluster and get a pointer to the correct typespec object cleanly and efficiently implementing the allocator becomes straight-forward. The important code is shown here:

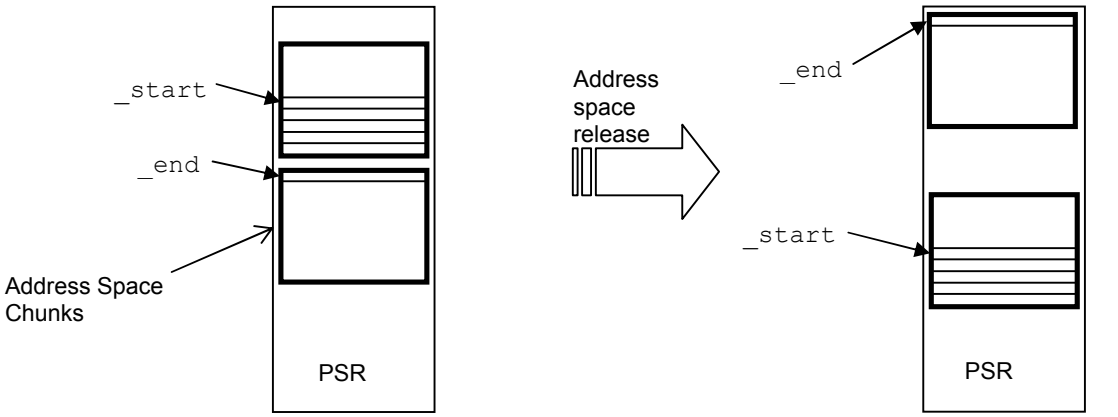
```
pointer allocate (size_type num, const void* = 0)
{
   //pointer ret = (pointer)(::operator new(num*sizeof(T)));

   int n = num+1;
   pointer ret = (pointer)(::operator new((n*sizeof(T)),
     os_cluster::of(this),
     ts< T >(), n));
   return ret;
}
```

Shown in comments above is the call to operator new normally employed by an STL allocator. It simply returns a pointer to heap allocated storage of the correct size. An ObjectStore allocator invokes a similar call but this time using the overloaded new provided by the ObjectStore library. `os_cluster::of(this)` is used to retrieve the cluster in which this allocator object resides, and because the allocator is provided as a template argument when the STL container is created, this will be the cluster which holds the container. A call to the `ts<T>()` template function returns a pointer to the correct typespec object.

The ObjectStore allocator is different in another way in that it returns memory bigger than requested. There are two related reasons for this. Internally, STL implementations often store pointers one-beyond-the-end of allocated arrays. These pointers are never de-referenced at runtime. They are used to for things like efficient pointer comparisons during iterator increment, but they do cause problems for the ObjectStore database verification tool osverifydb. These one-beyond-the-end pointers when stored persistently will be targeting either deleted or un-initialized storage, or storage of the incorrect type. The osverifydb utility does not know that these pointers are never de-referenced so it will flag them as data errors. This situation would make osverifydb less useful in guarding against real data errors.

The second reason is more serious. If, perchance, a persistently allocated vector was mapped so that it ended exactly on the last byte of an address space chunk, the one-beyond-the-end pointer would point into the beginning of the next chunk. In the event of address space release, ObjectStore does not associate the pointer with the vector and can relocate the chunk containing the vector *after* the pointer. The one-beyond-the-end pointer is no longer targeted one beyond the end of the vector. This is obviously erroneous and is illustrated below:



Address space generation ‘n’ Address space generation ‘n+1’

By allocating space for an extra T we solve both these problems. The one-beyond-the-end pointer is correctly targeted at initialized memory of the correct type, eliminating osverifydb errors, and because ObjectStore must maintain the invariant of contiguous allocation for C++ arrays, the extra slot in the vector can never be separated therefrom, so it can never exist in a separate address space chunk and the one-beyond-the-end pointer is guaranteed to remain as such.

Using the ObjectStore STL Allocator

Using the allocator to create persistent STL collections is now very simple. We discuss two examples here; one which embeds an STL list of Foo objects and a list iterator as a data member, the other which stores an STL set of pointers-to-Foo objects at global scope. We assume that our ObjectStore allocator is identical to the one described above and is a class named OTLAlloc

Example 1. Embedded STL Data member

First we show the pertinent parts of the header file showing the declaration of a persistent class A with embedded data members; a list of Foo objects and an iterator.

```
#include "otlalloc.h"
#include "Foo.h"
#include <list>

using namespace std;

class A
{
public:
   void addElement(const Foo& f)
   {
      _allFoos.push_back(f);
   }

private:
   typedef list<Foo, OTLAlloc<Foo> > FooObjList;
   typedef FooObjList::iterator FooObjListItr;

   FooObjList _allFoos;
   FooObjListItr _itr;
};
```

Here we show the corresponding schema file entry required for the class A and the STL internal objects that will be persistently allocated.

```
#include "A.h"
OS_MARK_SCHEMA_TYPE(A)
OS_MARK_SCHEMA_TYPE(_STL::_List_node<Foo>);
```

Notice that because the list is an embedded data member there is no requirement to mark the schema for the list type itself. Also, this list node class is specific to the STLport implementation. For other STL libraries it will be necessary to mark the schema for different internal types.

The addElement() method can accept references to *transient* Foo objects and insert these correctly into the collection because STL collections will *copy* the objects inserted into them. Copying will invoke the list's allocator which means that the Foo object will be located in the same ObjectStore cluster as the STL list; which in turn is the same cluster as the persistent instance of A. So we can write code like:

```
A* aPtr = // get pointer to our persistent 'A' object somehow
Foo f;    // Foo object created on stack
aPtr->addElement(f);
```

If the object pointed to by aPtr happens to be a transient instance then the allocator's call to `os_cluster::of(this)` will return the transient cluster and the copy operation will correctly result in a transient copy of Foo.

Example 2. Global STL Collection

Here we show the code for a transaction which creates a persistent STL set of pointers-to-Foo in its own database segment and attaches it to a root, creates a persistent Foo in a dedicated segment for the Foo objects and inserts a pointer to this Foo object into the set.

```
typedef set<Foo*, LtPtrFoo, OTLAlloc<Foo*> > FooPtrSet;

// Database pointer initialized elsewhere…
os_segment* seg = db->create_segment();
seg->set_comment("Set Segment");

FooPtrSet* fpsp = new(seg, ts< FooPtrSet >()) FooPtrSet;
os_database_root* root = db->create_root("Foo Pointer Set");
root->set_value(fpsp);

seg = db->create_segment();
seg->set_comment("Foo Object Segment");

Foo* fp = new(seg, ts< Foo >()) Foo;
fpsp->insert(fp);
```

Here we can simply locate the Foo objects in their own segment and store pointers to them in the set. This means that internally the STL is copying pointers not whole Foo objects and for some use-cases this arrangement will be more efficient. Here are the schema file entries:

```
#include "Foo.h"
OS_MARK_SCHEMA_TYPE(Foo)

#include <set>
typedef set<Foo*, LtPtrFoo, OTLAlloc<Foo*> > FooPtrSet;
OS_MARK_SCHEMA_TYPE(FooPtrSet);
OS_MARK_SCHEMA_TYPE(_STL::_Rb_tree_node<Foo*>);
```

It is necessary to mark the Foo objects themselves as they are directly allocated within the database. Optionally we can mark the schema type of the set using a typedef, (assume that the functor LTPtrFoo has

been defined in Foo.h.) and we must mark the schema for the STL internal objects used by the set, (these tree nodes are again particular to STLport).

STL and ObjectStore Soft-Pointers

As shown above when using the STL with ObjectStore we have several storage options: we can persistently store Foo objects directly in an STL container or we can store pointers to Foo objects. We can use STL containers as data members of other persistent classes or use them at global scope and attach them to a database root. In all these cases very large extents may exhaust address space. One possible solution to this is to store ObjectStore soft-pointers to Foo within an STL collection. ObjectStore collections are implemented using soft-pointers internally specifically to help control address space reservation for very large extents, and we can achieve the same using STL by storing soft-pointers to Foo in STL collections. An example of this is shown below. First we show the header file for a class B with an embedded hash-multimap data member of soft-pointers to Foo keyed on an integer.

```
// Local headers
#include "otlalloc.h"
#include "Foo.h"

// STL headers
//
#include <hash_map>

using namespace std;

typedef hash_multimap<int, os_soft_pointer<Foo>, hash<int>, equal_to<int>,
   OTLAlloc< os_soft_pointer<Foo> > > FooSoftPtrHashMultiMap;
typedef FooSoftPtrHashMultiMap::iterator FooSoftPtrHashMultiMapItr;

class B
{
public:

   // Ctor etc. omitted for brevity...

   void addElement(const Foo* f)
   {
      Foo* fPtr = const_cast<Foo*>(f);
      int key = fPtr->getId();
      _allFoos.insert(FooSoftPtrHashMultiMap::value_type(key, fPtr));
   }

   void rewindItr()
   {
      _itr = _allFoos.begin();
   }

   Foo* getNextFoo()
   {
      Foo* ret = (*_itr).second;
      if(_itr != _allFoos.end()){
         _itr++;       // Inc to next...
      }
      else{
         rewindItr();  // ... or wrap at end.
      }
      return ret;
   }

private:
```

```
   // Embedded hash-multimap of Foo soft pointers
   FooSoftPtrHashMultiMap _allFoos;
   FooSoftPtrHashMultiMapItr _itr;
};
```

Now we show the schema file entries for this class:

```
#include "Foo.h"
OS_MARK_SCHEMA_TYPE(Foo)

#include "B.h"
OS_MARK_SCHEMA_TYPE(B)
typedef _STL::_Hashtable_node<_STL::pair<int const, os_soft_pointer<Foo> > >
    FooSoftPtrHashNodeTypedef;
OS_MARK_SCHEMA_TYPE(FooSoftPtrHashNodeTypedef)
```

Notice we do not need to mark the `FooSoftPtrHashMultiMap` and `FooSoftPtrHashMultiMapItr` types because these data members are embedded. However, we need to typedef the STL internal hashtable node class so we can mark it correctly using the usual schema macro. Now we can use this class in the normal way, for example:

```
os_soft_pointer<B> sp = // get hold of a persistent 'B' object somehow...

OS_BEGIN_TXN(txn1, 0, os_transaction::update)
{
   // Re-target the embedded iterator onto the first Foo object
   sp->rewindItr();
}
OS_END_TXN(txn1)

for(int i=0; i<10; i++)
{
   OS_BEGIN_TXN(txn2, 0, os_transaction::update)
   {
      Foo* f = sp->getNextFoo();
      std::cout << "Txn " << i << ": " << *f << std::endl;
   }
   OS_END_TXN(txn2)
}
```

Here we exploit the persistent iterator embedded within the B class to print out the first 10 Foo objects held in its embedded hash-multimap, one per transaction. So generally soft-pointers can be used with STL collections like any other class. There are however, two STL collections where this causes problems; vectors and deques. The problem lies with how ObjectStore handles real C++ arrays of soft-pointers at runtime.

In the vector and deque types soft pointers may not be substituted for hard pointers. The reason is that the hardness of pointers in ObjectStore is controlled by the application schema of the executable accessing the database. More importantly, the schema controlling the hardness of a pointer must have an enclosing class, struct, or union type. Thus, any top level allocation of soft pointers outside of an enclosing class is not supported by ObjectStore. In the deque and vector containers, the elements of the container are internally allocated directly as arrays. If you wish to use soft pointers with these containers, then you must either use os_Reference<T> objects, or provide your own pointer class which either publicly inherits from `os_soft_pointer32<T>` and provides an appropriate set of operators to call up into the soft-pointer base class, or achieve the same by embedding a soft-pointer within a bespoke smart pointer class (see appendix). Below we provide another code snippet showing how to use embedded vectors of ObjectStore references:

```
// Local headers
#include "otlalloc.h"
#include "Foo.h"

// STL headers
//
#include <vector>

using namespace std;

// Cannot use os_soft_pointer here for complex reasons to do with schema.
// Use os_Reference which embeds an os_soft_pointer therein.
//
typedef vector< os_Reference<Foo>, OTLAlloc< os_Reference<Foo> > >
   FooOSRefVector;
typedef FooOSRefVector::iterator FooOSRefVectorItr;

class C
{
public:
   // Ctors omitted for brevity...

   // Updaters
   //
   void reserve(int num)
   {
      _allFoos.reserve(num);
   }

   void addElement(const Foo* f)
   {
      Foo* fPtr = const_cast<Foo*>(f);
      _allFoos.push_back(fPtr);
   }

private:

   // Embedded vector of Foo os_References
   FooOSRefVector _allFoos;
   FooOSRefVectorItr _itr;

};
```

And the schema file entries are:

```
#include "C.h"
OS_MARK_SCHEMA_TYPE(C)
OS_MARK_SCHEMA_TYPE(os_Reference<Foo>)
```

Now persistent class C can be used in a similar way to any other persistent class held in ObjectStore (see similar code examples above).

STL vs ObjectStore Collections

Now there is a choice between using STL collections and ObjectStore collections there is the inevitable comparison between the two and the question of when to use one rather than the other. The most pragmatic approach would use that which is most appropriate in any context; with the standards of suitability being completely defined by that context.

ObjectStore collections have been designed for use with very large extents. They are implemented internally to use soft-pointers throughout and supply query methods to extract interesting subsets of their contents in a single transaction, and they will automatically recycle address space while the query executes. Queries on large collections can be optimized by judicious use of indexes. These query results are then valid between transactions because the results are returned as a transient collection of soft-pointers onto a subset of the persistent elements in the original collection, making it very simple to iterate through them using a standard ObjectStore cursor. The ObjectStore collections have been designed to have reasonable inter-process concurrency behaviour. All this is untrue for STL, which itself has many positive aspects too numerous to enumerate here. Therefore it is probably best to view the ObjectStore collections and the STL library as complementary alternatives, both contributing to the programmer's toolkit of software engineering solutions.

Potential Pitfalls

In this section we examine some of the potential pitfalls of using STL in a persistent context, and describe some of the possible solutions to these problems.

Address Space Consumption

STL collections were designed with transient space in mind. They have not been designed with an understanding of how ObjectStore operates. This, combined with the observation that in most cases transient collections tend to be smaller than persistent collections, means that very large persistent STL collections can result in address space exhaustion.

The solutions available are similar to those where address space problems arise using bespoke data structures implemented using C++ pointers and arrays. The solution space can range from tweaking the storage model to store STL collections of soft-pointers (i.e. changing `vector<Foo>` or `vector<Foo*>` to `vector< os_soft_pointer<Foo> >`), through to a complete algorithm and architecture redesign with a different transaction structure, new data-structures to support the new algorithms and object re-clustering to optimize page fetch and locking behaviour. There is no one-size-fits-all solution to address space problems and each will require a tailored approach.

Performance Issues

Performance issues may arise unexpectedly is situations when page fetch is excessive. For example, some STL algorithms may fault in very many database pages and although the chosen algorithm would be efficient enough in transient memory, because of the requirement to fetch the pages containing the STL collection from the database, the performance is unacceptable in persistent memory. A contrived example might be sorting some collection. If a particular use-case required access to the elements in a particular order it might be tempting to simply sort the existing collection at the point of use. An alternative might be to amortize the cost of this sort across other use-cases and keep the collection sorted when the collection is updated, or change the collection used to better support the sort and perhaps compromise performance elsewhere.

In short, performance issues may arise using STL persistently, but these will be solved using the same techniques used for other optimization problems. Profile the code at runtime to identify where the time is being spent and which pages are being fetched. Develop a clear hypothesis as to the cause of the unwarranted behaviour. Design and implement a faster solution, and then confirm that it is faster and quantify by how much, with appropriate tests.

Inter-Process and Inter-Session Deadlocks

Deadlocks can arise between processes using STL if internally the collections take page locks in a different order from one another. If two processes access the same STL collection instance the order in which pages are locked as the STL code runs is entirely dependent on the data structures and access patterns invoked by the current use-case and how the STL internals are clustered across database pages by the STL allocator. Using MVCC readers, either sessions or processes, can help greatly with deadlock behaviour since these can neither be the cause of, or subject to deadlocks, but MVCC is not always appropriate. In these cases zero deadlock behaviour will be impossible to guarantee without the programmer acquiring page locks in a particular order, and the simplest way to do this is to use an implementation of the persistent process mutex pattern. An example of this which extends the example code above is shown here:

```
class WriteLocker
{
public:
   WriteLocker()
     :_exp(0)
   {}

   WriteLocker(void* p)
   {
       if(objectstore::is_persistent(p))
       {
          // Wait forever to get a write lock
          _exp = objectstore::aquire_lock(p, os_write_lock, -1);
       }
   }

   ~WriteLocker()
   {
     delete _exp;
     _exp=0;
   }

private:
   os_lock_timeout_exception* _exp;

};

class A
{
public:
   void addElement(const Foo& f)
   {
      WriteLocker(this);
      _allFoos.push_back(f);
   }

private:

   typedef list<Foo, OTLAlloc<Foo> > FooObjList;
   typedef FooObjList::iterator FooObjListItr;

   FooObjList _allFoos;
   FooObjListItr _itr;
};
```

Here we use a WriteLocker object that will wait indefinitely for a write lock on the page containing the instance of A. When this is achieved only then do we use the STL collection. If all the mutator functions in class A follow this protocol then no ObjectStore deadlocks will occur as a result of accessing or updating the embedded STL list

Consistency of STL Iterators

A related point is the consistency of STL iterators embedded in persistent classes such as the one above. The rules governing these are similar as for the STL itself. STL specifies the conditions under which certain types of iterators become unusable. All these conditions hold when iterators are stored persistently and it is the programmer's task to ensure that when the methods on persistent classes terminate all iterator data members are left in a consistent state.

There is also an issue with the consistency of transient STL iterators across transactions. In a situation where it is necessary to spin across a large STL collection in multiple transactions using a transient iterator onto a persistent STL collection it will be impossible to guarantee the consistency of this iterator between transactions. This situation is much the same as spinning a transient C++ pointer across a persistent C++ array over multiple transactions; it is possible for the array to be re-located to a different address in the PSR between transactions without the transient pointer being updated in sync. If this occurs, dereferencing the pointer will at best crash the program. Likewise, parts of the STL internals can be relocated in memory invalidating the transient STL iterator.

Multi-threading

The expectations of the MT behaviour of any particular STL implementation when used persistently will be no better than that guaranteed when it is used transiently. It will be the programmer's responsibility to control thread access to the persistent STL collections using appropriate synchronization objects such as mutexes and semaphores. This situation presents a level of complexity that would arise even if bespoke data structures using C++ pointers and arrays and as such, is independent of the STL. Have realistic expectations about the thread safety of STL containers! For more info see [4].

Transient STL Internal Objects

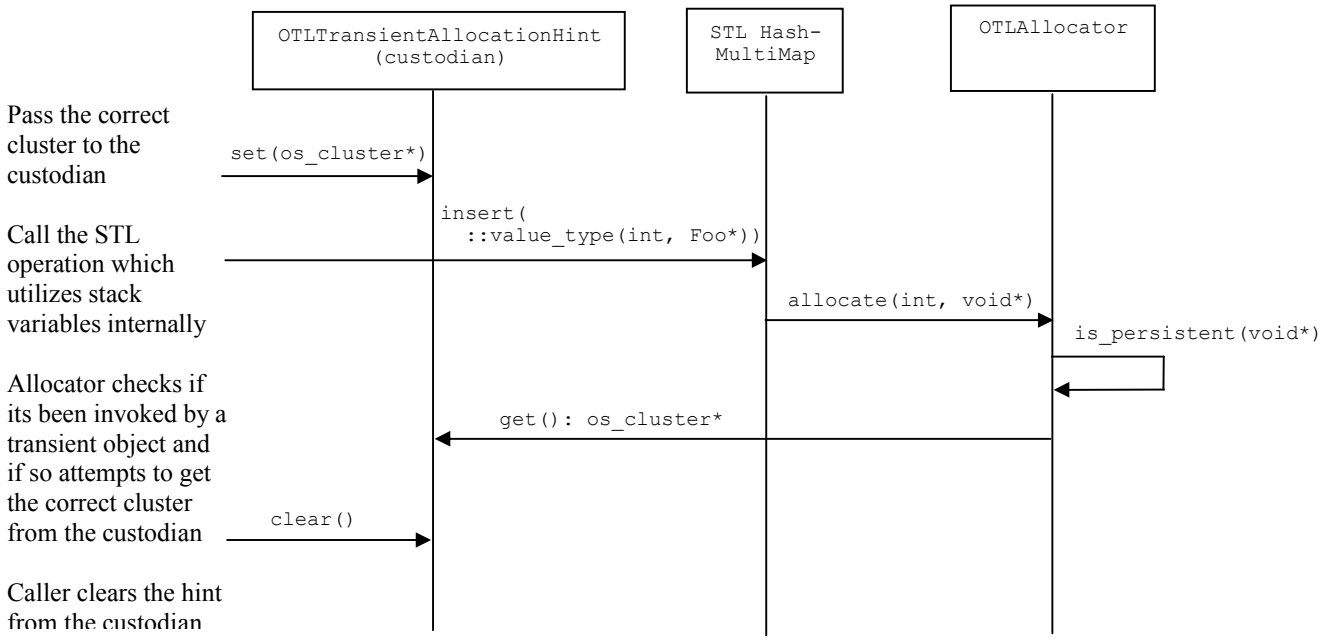
Unfortunately the preceding discussion has been somewhat simplified because it has omitted the problem of transient STL internal objects. During any operation, code internal to the STL may utilize a stack variable when appropriate, which in turn may itself allocate memory. Examples of where this occurs in STLport are:

1. Sorting an STL list with the dedicated list sort algorithm.
2. Resizing the number of buckets in the non-standard hash-based container classes. This can occur implicitly when inserting new elements.

These are examples of a number of STL operations which, in the absence of remedial action, will result in pointers to transient objects being created within persistent STL internal objects and the subsequent occurrence of ObjectStore exceptions (`err_unknown_pointer`) at transaction commit. These exceptions are not caused by 'impurities' within STLport's adherence to the allocator protocol, but by the interaction between STLport's quite acceptable use of stack based objects and the behaviour of the `os_cluster::of(this)` call used within our ObjectStore allocator.

The solution is to ensure that when STL employs a stack object during an operation on a persistent collection the allocator checks for this condition and has a reliable way of finding the correct ObjectStore cluster. Here we present a simple, reliable, universal and platform portable way of achieving this by employing a dedicated singleton which acts as a custodian of the correct cluster on behalf of the allocator, providing a global access point to this cluster when required by allocator.

Below we show an interaction diagram which describes the sequence of events used by this technique to pass the correct clustering hint into the allocator.



So that STL user needs to make two extra calls: set and clear. We can see from this that we need to provide the custodian class. A simple example is shown here:

```
class OTLTransientAllocationHint
{
public:
   static void set(os_database* db){
      assert(db);
      os_segment* seg = db->get_default_segment();
      _clr = seg->get_default_cluster();
   }

   // Assert calls omitted below for brevity
   static void set(os_segment* seg){_clr = seg->get_default_cluster();}
   static void set(os_cluster* clr){_clr = clr;}
   static void set(void* ptr) {_clr = os_cluster::of(ptr);}

   // If null is returned the allocator will use the transient cluster.
   static os_cluster* get(){return _clr;}
   static void clear(){_clr=0;}

private:
   static os_cluster* _clr;
};
```

There are several overloadings of the set() method so that databases, segment and clusters along with arbitrary pointers to persistent objects can be used to provide the appropriate hint. This class is implemented to hold an os_cluster pointer directly so we can provide these overloadings easily and avoid problems of pointers-to-persistent-objects becoming invalid between transactions; os_cluster pointers are transient and are valid as long as the database containing them is open. For particularly complex situations a more robust implementation of this class may employ a stack-like collection of os_cluster pointers so that different hint contexts could be properly maintained during execution. However, this class is sufficient for most purposes.

We also need to alter the allocator so that it checks for transient allocations. The most efficient way to achieve this is with a call to `objectstore::is_persistent(this)`. The adjusted code for the allocator is shown below:

```
pointer allocate (size_type num, const void* = 0)
{
   os_cluster* clr=0;
   if(objectstore::is_persistent(this))
   {
      clr = os_cluster::of(this);
   }
   else
   {
      clr = OTLTransientAllocationHint::get();
      if(!clr)
      {
         clr = os_cluster::get_transient_cluster();
      }
   }
   assert(clr);

   int n = num+1;
   pointer ret = (pointer)(::operator new((n*sizeof(T)),
      clr, ts< T >(), n));

   return ret;
}
```

There are three possible code paths here. First, the allocator always uses the cluster-of-this if its associated with a persistent object ('this' is persistent). Secondly, if the allocator is associated with a transient object ('this' is transient) and the call to the custodian returns a valid (non-null) cluster then the allocator will use it. This case occurs when the STL is allocating stack objects internally while operating on a persistent STL container. Thirdly, if 'this' is transient and the call to the custodian returns a null pointer, then it is assumed that this *really is* a transient STL object and the allocator uses the transient cluster. This case can occur for example, when a transient class has a data member which is an STL type that uses the ObjectStore allocator. Here is some code by way of clarification:

```
// Define a persistent version of the STL basic string
typedef std::basic_string<char, std::char_traits<char>, OTLAlloc<char> >
   OTLString;

class Foo
{
public:
   // Most code omitted for brevity...

private:
   int _id;
   // Bury a persistent STL string here as a data member
   OTLString _name;
};
```

If we create a Foo *transiently*, for example on the stack, then at some point the construction of _name will invoke our allocator and the third case described above will execute.

Aside from ensuring that no transient pointers are allocated by the interaction between our allocator and the STL we must ensure that clustering is not compromised. Specifically:

1. Persistent STL internal objects are clustered correctly with the other component parts of their container. It is less than optimal to have STL internal objects scattered about the database at random.
2. The clusters returned from the custodian are in databases that are currently open.
3. Genuine transient STL allocations do not result in part of their internals being allocated persistently. This will result in orphaned objects within the database.

These are all achieved by setting and clearing the correct clustering hint around any STL call that internally utilizes stack objects. The custodian class OTLTransientAllocationHint shown above tracks a single cluster pointer, but in situations where deeply nested calls occur this may prove insufficient. On the contrary, it may be necessary to maintain a stack-like collection of cluster pointers so that as the program context alters, the cluster pointer at the top of the stack is always correct for the STL operation in the current context. One of the difficulties here is ensuring that the stack of os_cluster pointers is correctly unwound as execution proceeds, and use of a variant of the scoped locking idiom [5] can assist with this. An example is shown below:

```
class OTLTransientObjectGuard
{
public:
   OTLTransientObjectGuard(os_database* db) {
      OTLTransientAllocationHint::set(db);}

   OTLTransientObjectGuard(os_segment* seg) {
      OTLTransientAllocationHint::set(seg);}

   OTLTransientObjectGuard(os_cluster* clr) {
      OTLTransientAllocationHint::set(clr);}

   OTLTransientObjectGuard(void* ptr) {
      OTLTransientAllocationHint::set(ptr);}

   ~OTLTransientObjectGuard(){
      OTLTransientAllocationHint::clear();}
};
```

This guard class interfaces with our custodian class and can be used to ensure that upon exit from a particular block of code the caller can guarantee that the hint held by the custodian is cleared, even in the event of exceptions. In an implementation employing a stack of cluster pointers it would pop the stack.

Now we show how consideration for stack allocated STL internal objects impacts the calling code. Our example demonstrates the changes necessary for the code in class B shown earlier that inserts into a non-standard hash-based container:

```
typedef hash_multimap<int, os_soft_pointer<Foo>, hash<int>, equal_to<int>,
   OTLAlloc< os_soft_pointer<Foo> > > FooSoftPtrHashMultiMap;
typedef FooSoftPtrHashMultiMap::iterator FooSoftPtrHashMultiMapItr;

class B
{
public:
   void addElement(const Foo* f)
   {
      OTLTransientObjectGuard guard(this);

      Foo* fPtr = const_cast<Foo*>(f);
      int key = fPtr->getId();
      _allFoos.insert(FooSoftPtrHashMultiMap::value_type(key, fPtr));
   }

private:
```

```
    // Embedded hash-multimap of Foo soft pointers
    FooSoftPtrHashMultiMap _allFoos;
    FooSoftPtrHashMultiMapItr _itr;
};
```

Inserting into a hash-multimap can cause it to resize. When it resizes STLport uses a temporary stack variable so we protect against transient pointer errors by constructing a guard object passing in 'this' as the hint. Now the scenario described in the interaction diagram above unfolds and when the guard object goes out of scope the custodian is cleared.

Using our guard class results in a single extra line of code for the STL user. In situations where the cost of construction cannot be tolerated, direct calls into OTLTransientAllocationHint can be substituted. This technique is simple, reliable, universal and works for every case where STL employs stack variables internally.

Conclusion

This paper has several aims. It demonstrates how easily STL objects can be stored persistently within ObjectStore using a standard STL allocator. A fully implemented ObjectStore allocator template OTLAlloc is presented that works for the popular STLport implementation, along with the associated classes to construct the necessary os_typespec objects and locate the appropriate ObjectStore clusters. We outline some of the caveats when using STL persistently such as: address space consumption, runtime performance, deadlocks, iterator consistency and multi-threading. Finally we present a simple, reliable and universal technique for avoiding problems caused by STL's internal use of temporary stack objects. The code examples are all taken from working code compiled and run on Microsoft Visual C++ v6 that has been proven to scale to persistent collection sizes in excess of 3 million elements.

The most important aim of this paper is to engender the idea that storing STL persistently, directly in a database that offers full transaction semantics and concurrency control, such as ObjectStore, can revolutionize the way we conceptualize and implement data storage and data access in C++.

Appendix

ObjectStore STL Allocator Code

This code is an implementation that works with STLport and the Microsoft VC++ compiler. ObjectStore specific code is in bold.

```
#ifndef OTLALLOC_H
#define OTLALLOC_H

// ObjectStore Headers
#include <ostore\ostore.hh>

// OTL Header - see below
#include "otlts.h"

// STL headers
#include <limits>

/**
 * Implementation of a standard STL Allocator which allows STLport
 * containers to be stored within ObjectStore
 */
template <class T>
class OTLAlloc
{
public:

   // Type definitions
   typedef T        value_type;
   typedef T*       pointer;
   typedef const T* const_pointer;
   typedef T&       reference;
   typedef const T& const_reference;
   typedef std::size_t    size_type;
   typedef std::ptrdiff_t difference_type;

   // Return address
   //
   pointer address (reference x) const
   {
      return &x;
   }

   const_pointer address (const_reference x) const
   {
      return &x;
   }

   // Standard Ctors and a dtor
   //
   OTLAlloc() throw()
   {}

   // Rebind allocator to type U
   //
   template <class U>
   struct rebind
   {
      typedef OTLAlloc<U> other;
```

```
};

template <class U>
OTLAlloc (const OTLAlloc<U>&) throw()
{}

OTLAlloc(const OTLAlloc&) throw()
{}

~OTLAlloc() throw()
{}

// Return maximum number of elements that can be allocated
//
size_type max_size () const throw()
{
   return std::numeric_limits<std::size_t>::max() / sizeof(T);
}

// Allocate but don't initialize num elements of type T
//
pointer allocate (size_type num, const void* = 0)
{
   os_cluster* clr=0;
   if(objectstore::is_persistent(this))
   {
      clr = os_cluster::of(this);
   }
   else
   {
      clr = OTLTransientAllocationHint::get();
      if(!clr)
      {
         clr = os_cluster::get_transient_cluster();
      }
   }
   assert(clr);

   int n = num+1;
   pointer ret = (pointer)(::operator new((n*sizeof(T)),
      clr, ts< T >(), n));

   return ret;
}

// Initialize elements of allocated storage p with value value
//
void construct (pointer p, const T& value)
{
   // Initialize memory with placement new
   new((void*)p)T(value);
}

// Destroy elements of initialized storage p
void destroy (pointer p)
{
   // Destroy objects by calling their destructor
   p->~T();
}
```

```
   // Deallocate storage p of deleted elements
   //
   void deallocate (pointer p, size_type num)
   {
      ::operator delete((void*)p);
   }
};

// Return that specifies all specializations of this allocator
// are interchangeable (i.e. the allocators do not hold state).
//
template <class T1, class T2>
bool operator== (const OTLAlloc<T1>&, const OTLAlloc<T2>&) throw()
{
   return true;
}


template <class T1, class T2>
bool operator!= (const OTLAlloc<T1>&, const OTLAlloc<T2>&) throw()
{
   return false;
}


/**
 * Extra template function definition within the std namespace required
 * when using stlport with MSVC6. This is necessary because the
 * compiler does not support member template classes.
 */
namespace std
{

template <class T, class U>
OTLAlloc<U>& __stl_alloc_rebind(OTLAlloc<T>& a, const U*)
{
   return (OTLAlloc<U>&)(a);
}

}

#endif
```

Typespec Template Code

Here we show the full code listing for the templates and template functions that enable ObjectStore typespecs to be efficiently accessed from anywhere, but particularly from inside other templates.

```
#ifndef OTLTS_H
#define OTLTS_H

// Local headers
#include "otlutil.h"

// System headers

// ObjectStore headers
#include <ostore/ostore.hh>

// Template which improves the syntax for T::get_os_typepsec().
// User defined classes do not need to declare the static
// get_os_typepsec() method. Uses Singleton pattern for efficiency,
// so that repeated calls access cached os_typespecs. Each call to
// these classes will instantiate a new instance of the os_typespec
// on the heap. If its already been allocated for this type then it
// is simply returned.
//
template <class T>
class TSObj
{
public:

   static os_typespec* getType(T* dummy=0)
   {
      // Use Singleton to see if we have already got the
      // os_typespec for this type.
      if(_type==0)
      {
         init();
      }

      return _type;
   }

   // Singleton initialisation in separate public method
   // should it be necessary for MultiThreaded programs
   // to remove the need to take mutex locks for performance
   // reasons.
   //
   static void init()
   {
      // Get a typespec-compatible class name from
      // NameMaker template (via template function).
      const char* typeStr = getClassName< T >();

      // Actually get the typespec for the first time.
      // This will work with templated types, pointers,
      // native types and typedefs.
      _type = new os_typespec(typeStr);

      // Clean up. Does not require deletion since these
      // chars are still 'owned' by NameMaker.
      typeStr=0;
   }
```

```
private:

   // Plays the role of _instance in a GOF Singleton.
   // Declare volatile so that MT code that requires the
   // double-checked-locking pattern will work
   static os_typespec* volatile _type;
};

// Static initialisers
template < class T> os_typespec* volatile TSObj< T >::_type=0;

// This strategy for finding typespecs uses a templated singleton
// technique. Does not require static get_os_typepsec's to be defined
// within user classes.  Works with classes, templated types
// (e.g. collections), native types, typedefs, structs and unions and
// pointers to any of these. Also works with dictionaries and any
// multi-argument template.
//
// The dummy argument is required to fixup a horrible bug in
// Microsoft V6 implementation of no-arg function templates.
//
template<class T>
os_typespec* ts(T* dummy=0)
{
   return TSObj< T >::getType();
}

#endif
```

NameMaker Template Code

```
#ifndef OTLUTIL_H
#define OTLUTIL_H

// System headers
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include <typeinfo.h>
#include <assert.h>

// ObjectStore headers
#include <ostore\ostore.hh>

// Template that returns the name of the class T but normalized
// so that mulitple whitespace characters are reduced to a single space
// and preceding 'class' and 'struct' specifiers are stripped. Uses a
// static singleton for two reasons: so that callers do not have to
// delete the returned string, and for speed; after the first call it
// is very fast.
//
template<class T>
class NameMaker
{
public:

   static const char* makeName(T* dummy=0)
   {
      if(!_name)
      {
         init();
      }
      return _name;
   }

   // Singleton initialization in separate public method
   // should it be necessary for MultiThreaded programs
   // to remove the need to take mutex locks for performance
   // reasons.
   static void init()
   {
      // Get a copy of the class name as a string...
      const char* typeName = typeid(T).name();
      _name = new char[strlen(typeName)+1];
      strcpy(_name, typeName);
      // ..clean it (remove spurious spaces etc).
      cleanName(_name);
   }

private:

   // Protected helper function that reduces all white space to
   // a single white space character and strips unnecessary prefixes
   // from the string
   //
   static void cleanName(char* s)
   {
      // Read and write pointers
      char *rp, *wp;
```

```
// Remove the 'class' prefix from rtti discovered class name
// should it be necessary.
if(!strncmp(s, "class ", 6))
{
   wp = s;
   rp = s+6;
}
else if(!strncmp(s, "struct ", 7))
{
   wp = s;
   rp = s+7;
}
else if(!strncmp(s, "union ", 6))
{
   wp = s;
   rp = s+6;
}
else
{
   // Point them at the char array...
   rp = wp = s;
}

// While we've got chars here..
while(*rp)
{
   // Copy all the chars...
   if(!isspace(*rp))
   {
      *wp = *rp;
      wp++;
   }
   else if ( (rp>s) && (!isspace(*(rp-1))) && (*(rp+1)!='*') )
   {
      *wp = ' ';
      wp++;
   }

   // Read the next char...
   rp++;

   // Skip the word 'class' or struct if found...
   // but only if its not part of a class or struct name
   if(!strncmp(rp, "class", 5) && !isCSym(*(rp-1)) &&
      !isCSym(*(rp+5)))
   {
      rp += 5;
   }
   else if(!strncmp(rp, "struct", 6) && !isCSym(*(rp-1)) &&
      !isCSym(*(rp+6)))
   {
      rp += 6;
   }
   else if(!strncmp(rp, "union", 5) && !isCSym(*(rp-1)) &&
      !isCSym(*(rp+5)))
   {
      rp += 5;
   }

}

if(!isspace(*(wp-1)))
{
```

```
         *wp = '\0';
      }
      else
      {
         *(wp-1) ='\0';
      }

   }

   // Returns true if the passed character could legally be used
   // in a C++ class or struct name. i.e. alphanumeric or underscore.
   static bool isCSym(char c)
   {
      return (isalnum(c) || c == '_');
   }

   // *** Implementation ***

   // Plays the role of _instance in a GOF Singleton.
   static char* _name;
};

// Static initialisers
template < class T> char* NameMaker< T >::_name=0;


// The dummy argument is required to fixup a horrible bug in
// Microsoft implementation of no-arg function templates.
//
template<class T>
const char* getClassName(T* dummy=0)
{
   return NameMaker< T >::makeName();
}

#endif
```

Pointer Class that Inherits from ObjectStore Soft-Pointer

Here we show an implementation of a pointer class which inherits from the ObjectStore soft-pointer class which will work with STL vectors and deques.

```
template<class T>
class softptr : public os_soft_pointer32<T>
{
private:
   typedef os_soft_pointer32<T> baseptr;

public:
   static os_typespec* get_os_typespec();

   softptr() {}
   softptr(T* p) : os_soft_pointer32<T>(p) {}
   softptr(softptr<T> const& p) : os_soft_pointer32<T>((baseptr const&)p)
   {}

   softptr<T>& operator=(T* p) { baseptr::operator=(p); return *this; }
   softptr<T>& operator=(softptr<T> const& sp) {
      baseptr::operator=((baseptr const&)sp); return *this;}

   os_boolean operator==(T const* p) const {
      return baseptr::operator==(p); }
   os_boolean operator==(softptr<T> const& sp) const {
      return baseptr::operator==((baseptr const &)sp); }

   os_boolean operator!=(T const* other) const {
     return !(*this == other);}
   os_boolean operator!=(softptr<T> const& other) const {
     return !(*this == other); }

   os_boolean operator<(T const* p) const {
     return baseptr::operator<(p); }
   os_boolean operator<(softptr<T> const& other) const {
     return baseptr::operator<((baseptr const&)other); }

   os_boolean operator>(T const* p) const {
     return baseptr::operator>(p); }
   os_boolean operator>(softptr<T> const& other) const {
     return baseptr::operator>((baseptr const&)other); }

   os_boolean operator<=(T const* other) const {
     return !(*this > other); }
   os_boolean operator<=(softptr<T> const& other) const {
     return !(*this > other); }

   os_boolean operator>=(T const* other) const {
     return !(*this < other); }
   os_boolean operator>=(softptr<T> const& other) const {
     return !(*this < other); }
};
```

Pointer Class that Wraps an ObjectStore Soft-Pointer

Here we show an implementation of a pointer class which wraps an ObjectStore soft-pointer so it can work with STL vectors and deques. Aggregation has an advantage over inheritance in that a properly designed soft pointer wrapper class still allows for schema controlled hardness of pointers.

```
// Use this preprocessor macro.
#if defined(USE_HARD_POINTERS)
#define PTR_TYPE(A) A*
#else
#define PTR_TYPE(A) os_soft_pointer<A>
#endif

template<class T>
class stlport_soft_pointer_wrapper
{
private:
  PTR_TYPE(T) ptr_;

public:
   stlport_soft_pointer_wrapper() : ptr_(0) {}
   stlport_soft_pointer_wrapper(T* p) : ptr_(p) {}
   stlport_soft_pointer_wrapper(const stlport_soft_pointer_wrapper<T>& p)
      : ptr_((T*)p) {}
   stlport_soft_pointer_wrapper(os_soft_pointer<T>& p) : ptr_((T*)p) {}

   operator T*() const { return (T*)ptr_; }
   T* operator->() const { return (T*)ptr_; }

   stlport_soft_pointer_wrapper& operator=(T* p)
   { ptr_ = p; return *this; }
   stlport_soft_pointer_wrapper& operator=
      (const stlport_soft_pointer_wrapper<T>& p)
   { ptr_ = p; return *this; }
   stlport_soft_pointer_wrapper& operator=(os_soft_pointer<T> const& sp)
   { ptr_ = (T*)sp; return *this; }

   os_boolean operator==(T const* p) const
   { return ptr_ == p; }
   os_boolean operator==(stlport_soft_pointer_wrapper<T> const& p) const
   { return ptr_ == p; }
   os_boolean operator==(os_soft_pointer<T> const& p) const
   { return p == ptr_; }

   os_boolean operator!=(T const* p) const
   { return !(*this == p); }
   os_boolean operator!=(stlport_soft_pointer_wrapper<T> const& p) const
   { return !(*this == p); }
   os_boolean operator!=(os_soft_pointer<T> const& p) const
   { return !(*this == p); }

   os_boolean operator<(T const* p) const
   { return ptr_ < p; }
   os_boolean operator<(stlport_soft_pointer_wrapper<T> const& p) const
   { return ptr_ < (T*)p; }
   os_boolean operator<(os_soft_pointer<T> const& p) const
   { return ptr_ < (T*)p; }
```

```
    os_boolean operator>(T const* p) const
    { return ptr_ > p; }
    os_boolean operator>(stlport_soft_pointer_wrapper<T> const& p) const
    { return ptr_ > (T*)p; }
    os_boolean operator>(os_soft_pointer<T> const& p) const
    { return ptr_ > (T*)p; }

    os_boolean operator<=(T const* p) const
    { return !(*this > p); }
    os_boolean operator<=(stlport_soft_pointer_wrapper<T> const& p) const
    { return !(*this > p); }
    os_boolean operator<=(os_soft_pointer<T> const& p) const
    { return !(*this > p); }

    os_boolean operator>=(T const* p) const
    { return !(*this < p); }
    os_boolean operator>=(stlport_soft_pointer_wrapper<T> const& p) const
    { return !(*this > p); }
    os_boolean operator>=(os_soft_pointer<T> const& p) const
    { return !(*this > p); }
};
```

Macros to Simplify Schema Marking

Here we show some schema macros which simplify marking the STLport internal types in the schema file. These should be placed in a header file and made available using #include in the schema file.

```
#if !defined(OS_MARK_STLPORT_VECTOR)
#define OS_MARK_STLPORT_VECTOR(C,ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((vector < C, ALLOC >))
#endif

#if !defined(OS_MARK_STLPORT_DEQUE)
#define OS_MARK_STLPORT_DEQUE(C,ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((deque < C , ALLOC >))
#endif

#if !defined(OS_MARK_STLPORT_LIST)
#define OS_MARK_STLPORT_LIST(C, ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((list < C , ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_List_node< C >))
#endif

#if !defined(OS_MARK_STLPORT_SET)
#define OS_MARK_STLPORT_SET(C,ORDER,ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((set< C, ORDER, ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Rb_tree_node< C >))
#endif

#if !defined(OS_MARK_STLPORT_MAP)
#define OS_MARK_STLPORT_MAP(KEY, DATA, ORDER, ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((map< KEY, DATA, ORDER, ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Rb_tree_node< _STL::pair< KEY const, DATA > >))
#endif

#if !defined(OS_MARK_STLPORT_MULTISET)
#define OS_MARK_STLPORT_MULTISET(KEY, ORDER, ALLOC) \
OS_MARK_SCHEMA_TYPESPEC((multiset< KEY , ORDER , ALLOC > )); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Rb_tree_node< KEY >));
#endif

#if !defined(OS_MARK_STLPORT_MULTIMAP)
#define OS_MARK_STLPORT_MULTIMAP(KEY, DATA, ORDER, ALLOC)
OS_MARK_SCHEMA_TYPESPEC((multimap< KEY , DATA , ORDER , ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Rb_tree_node< _STL::pair< KEY const, DATA >
>));
#endif

#if !defined(OS_MARK_STLPORT_HASH_SET)
#define OS_MARK_STLPORT_HASH_SET(KEY, HASHFCN, EQUALKEY, ALLOC)
OS_MARK_SCHEMA_TYPESPEC((hash_set< KEY , HASHFCN , EQUALKEY , ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Hashtable_node< KEY >));
#endif

#if !defined(OS_MARK_STLPORT_HASH_MAP)
#define OS_MARK_STLPORT_HASH_MAP(KEY, DATA, HASHFCN, EQUALKEY, ALLOC)
OS_MARK_SCHEMA_TYPESPEC((hash_map< KEY, DATA, HASHFCN, EQUALKEY, ALLOC >)); \
OS_MARK_SCHEMA_TYPESPEC((_STL::_Hashtable_node<_STL::pair< KEY const, DATA >
>));
#endif
```

```
#if !defined(OS_MARK_STLPORT_HASH_MULTIMAP)
#define OS_MARK_STLPORT_HASH_MULTIMAP(KEY, DATA, HASHFCN, EQUALKEY, ALLOC)
OS_MARK_SCHEMA_TYPESPEC((hash_multimap< KEY, DATA, HASHFCN, EQUALKEY, ALLOC >))
#endif

#if !defined(OS_MARK_STLPORT_HASH_MULTISET)
#define OS_MARK_STLPORT_HASH_MULTISET(KEY, HASHFCN, EQUALKEY, ALLOC)
OS_MARK_SCHEMA_TYPESPEC((hash_multiset< KEY, HASHFCN, EQUALKEY, ALLOC >));
#endif
```

---

[1] ObjectStore Bookshelf, http://support.exln.com/i/documentation/doc_ostore.asp

[2] Nicolai M. Josuttis, "User-Defined Allocator", http://www.josuttis.com/cppcode/allocator.html.

[3] E.Gamma, R.Helm, R.Johnson, J.Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[4] Scott Meyers, Effective STL, Addison-Wesley, 2001 ISBN 0-201-74962-9

[5] Scoped Locking Idiom, D.C.Scmidt, M. Stal, H. Rohnert and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects Vol 2. Willey & Sons, New York, 2000