DØOM Design and Implementation

scott snyder

Release Overview

- `CINT`
  - Latest release of cint, via CERN.
  - Numerous local changes, additions.
    - `lite`: To build only the pieces of the library needed for dictionary support.
    - `CINT/CINT/d0cintinc`: Versions of the standard library headers to be read by `d0cint`. Many define only the types and omit operations.
- `d0om`
  - `d0om` — Headers, inline implementations, template implementations, and make scraps.
  - `d0om/tests` — Headers for test programs.
  - `doc` — User guide.
  - `scripts` — Release-related shell scripts.
  - `src/common` — libd0om.
  - `src/d0cint` — `d0cint`-specific code.
  - `src/stream` — libstream.
  - `tests` — Test programs.
  - `ups` — Standard UPS setup directory.

Header Summary

- Utilities.

d0om_util.hpp	d0om_includes.hpp
d0om_auto_ptr.hpp	d0om_iosfwd.hpp
d0_String.hpp	d0_Refcount.hpp
d0om_Base_Alloc.hpp	d0om_Alloc.hpp
d0om_Hashmap.hpp	d0om_Ptrmap.hpp
d0om_Ptrrep_Hash.hpp	

- Error handling.

d0om_exceptions.hpp

- d0_Collection.

d0_Collection.hpp	d0_Collection_Base.hpp
d0_List.hpp	d0_List_1.hpp
d0_Map.hpp	d0_Map_1.hpp
d0_Multimap.hpp	d0_Multimap_1.hpp
d0_Multiset.hpp	d0_Multiset_1.hpp
d0_Set.hpp	d0_Set_1.hpp
d0_Vector.hpp	d0_Vector_1.hpp
d0_Iterator.hpp	
d0om_STL_Const_Iterator.hpp	
d0om_STL_Iterator.hpp	
d0om_STL_Replaceable_Iterator.hpp	
d0om_Collection_Iterator.hpp	

Header Summary

- Pointer Collections.

d0_PCollection.hpp	d0_PIterator.hpp
d0_PList.hpp	d0_PVector.hpp

- d0_Object and d0_Ref.

d0_Object.hpp	d0_Const_Object.hpp
d0_Ref.hpp	d0_Ref_Base.hpp
d0om_Indptr.hpp	d0om_Transient_Indptr.hpp
d0om_Registerable.hpp	d0om_Object_Registry.hpp
d0om_Object_Set.hpp	

- Reference Collections.

d0_RCollection.hpp	d0_RIterator.hpp
d0_RList.hpp	d0_RVector.hpp

- Output Filtering.

d0_Output_Filter.hpp
d0_Output_Filter_Base.hpp

- Stream.

d0Key.hpp	d0Stream.hpp
d0StreamDB.hpp	d0StreamFactory.hpp
d0StreamName.hpp	d0StreamRegistration.hpp

Header Summary

- Dictionary.

```
d0om_Memvar_Info.hpp
d0om_Type.hpp
d0om_Type_Atomic.hpp
d0om_Type_Class.hpp
d0om_Type_Classrep.hpp
d0om_Type_Collection.hpp
d0om_Type_Object.hpp
d0om_Type_Reference.hpp
d0om_d0_Collection_Adapter.hpp
d0om_Collection_Adapter.hpp
d0om_Collection_Adapter_Registry.hpp
d0om_STL_Assoc_Adapter.hpp
d0om_STL_Sequence_Adapter.hpp
```

- `CINT` Interface.

```
d0om_Cint_Interface.hpp
```

- Obsolete.

```
d0_basic_types.hpp
```

Object Definition

- Example:

```
class Mytype
{
private:
  int x1;
};

class Obj : d0_Object
{
private:
  int a;
  d0_String b;

  d0_Ref<Obj> c;
  Obj* cp;

  d0_Vector<float> d;
  vector<float> ds;

  Mytype f;
  double g;  // ! This field is not persistent.
  D0_OBJECT_SETUP (Obj);
};
```

Persistent Data Members

- Fundamental types: `int`, `short`, `long`, `char`, `bool`, `float`, `double`.
- `string` and `d0_String`.
- Enumeration types.
- Previously declared structures or classs.
- References: `d0_Ref` or C++ pointers (or `auto_ptr`).
  - Target type must derive from `d0_Object`.
  - Same object cannot be pointed to by both `d0_Ref` and C++ pointer.
- DØOM Collections: `d0_List`, `d0_Vector`, `d0_Set`, `d0_Multiset`, `d0_Map`, `d0_Multimap`.
- STL Collections: `list`, `vector`, `set`, `multiset`, `map`, `multimap`, `deque`.
- User-defined collections.

Other Features

- Can use typedef names, both global and class-local.
  - (Some known problems in the current release are fixed in the head version.)

- Namespaces partially supported.
  - Namespace using directives and declarations not supported.

- Use `#ifndef __CINT__` to hide something from `d0cint`.

- Can mark a data member as non-persistent by following it with a comment starting `//!`. (Same convention as ROOT.)

- For user-defined template classes, must explicitly declare the instantiations to be used:

  ```
  //! +class my_template_class<int>;
  ```

  This is not needed to use the collection and reference classes provided.

Restrictions

- No C++ references.
- No bitfields.
- No unions.
- No class can appear more than once in an inheritance lattice.
- Virtual base classes cannot contain persistent data.
- Targets of pointers must derive from `d0_Object`.
- Classes deriving from `d0_Object` should include `D0_OBJECT_SETUP(class)`.

Collections

User-Defined Collections

- A class is considered a collection if it defines the typedef name `d0om_collection_adapter`.
- A collection must also define the name `value_type`, as in STL.
- `d0om_collection_adapter` must name a class deriving from `d0om_Collection_Adapter`.
- Three concrete adapter instances are provided:
  - `d0om_STL_Assoc_Adapter<C>` - For STL associative container classes.
  - `d0om_STL_Sequence_Adapter<C>` - For STL sequence containers.
  - `d0om_d0_Collection_Adapter` - For `d0_Collection` classes.

User-Defined Collections

```
class d0om_Collection_Adapter
//
// Purpose: This defines this abstract interface we need to access
//          collection classes.  There will be concrete subclasses
//          for the various collection types.
//
{
public:
  // A concrete adapter class must implement the following five
  // functions.
  // The number of elements in the collection.
  virtual int size () const = 0;

  // The size of the collection instance, in bytes.
  virtual int collsize () const = 0;

  // Loop using CI, inserting elements into the collection.
  virtual void insert_elements (d0om_Collection_Iterator& ci) = 0;

  // Loop over collection elements using CI.
  virtual void iterate (d0om_Collection_Iterator& ci) = 0;

  // Make a new instance of the adapter, associated with a particular
  // collection instance COLL.
  //static d0om_Collection_Adapter* make (void* coll);

  // Allocate and deallocate collection adapters using d0om_Base_Alloc.
  void* operator new (size_t size);
  void operator delete (void* p, size_t size);
};
```

User-Defined Collections

```
class d0om_Collection_Iterator
//
// Purpose: Callback interface for iterating through collections.
//
{
public:
  // Constructor, destructor.
  d0om_Collection_Iterator () {}
  virtual ~d0om_Collection_Iterator () {}

  // How many elements remain to be processed?
  virtual int nleft () const = 0;

  // Called for each element.
  virtual void element (void* elt) = 0;
};
```

`d0_Collection` Types

- Available:
  - `d0_List<T>`
  - `d0_Vector<T>`
  - `d0_Set<T>`
  - `d0_Multiset<T>`
  - `d0_Map<K, T>`
  - `d0_Multimap<K, T>`
- All derive from the corresponding STL class.
- All also derive from `d0_Collection<T>`, which provides common ODMG-style operations.
- ODMG-style `d0_Iterator<T>`, created by `d0_begin` and `d0_end` methods. Also `d0_Const_Iterator<T>`.
- Also `d0_PList<T>` and `d0_PVector<T>`.
  - `d0_PList<T>` works like `d0_List<T*>`.
  - Derives from and shares code with `list<void const*>`.
- Also `d0_RList<T>` and `d0_RVector<T>`.
  - `d0_RList<T>` works like `d0_List<d0_Ref<T> >`.
  - Derives from and shares code with `list<d0_Ref<const d0_Object> >`.

d0_Collection_Base

```
class d0_Collection_Base
//
// Purpose: Abstract base class for all collections.
//
{
public:
  // Constructor, destructor.
  d0_Collection_Base () {}
  virtual ~d0_Collection_Base () {}

  // Retrieve static information about the type of collection this is.
  virtual bool is_ordered () const = 0;
  virtual bool allows_duplicates () const = 0;

  // sizeof the actual class of this object.
  virtual int  collsize () const = 0;

  // Container operations which don't depend on the element type.
  virtual int cardinality () const = 0;
  virtual bool is_empty () const = 0;
  virtual void remove_all () = 0;

  // Operations specifying elements as void*'s.
  // Use of these methods outside of the persistence mechanism
  //is discouraged.
  virtual void insert_element_ptr (const void* elem) = 0;
  virtual void remove_element_ptr (const void* elem) = 0;
  virtual bool contains_element_ptr (const void* elem) const = 0;

  // Special method used to insert a bunch of elements
  // into a collection.
  // Used by the persistence mechanism.
  // Will call inserter.element() each time passing it a void*
  // to an element to be filled in.  The method inserter.nleft()
  // controls how many elements we insert.
  virtual void insert_elements (d0om_Collection_Iterator& inserter) = 0;
```

d0_Collection_Base

```
  // Base iterator creation.
  // Use of this method outside of the persistence mechanism
  // is discouraged.
  virtual d0_Iterator_Base* create_new_iterator () const = 0;

protected:
  // Make these protected to disallow assignment
  // between Collection_Base's.
  d0_Collection_Base (const d0_Collection_Base& /*from*/);
  d0_Collection_Base& operator= (const d0_Collection_Base& /*from*/);
};
```

d0_Collection

```
template <class T>
class d0_Collection
  : virtual public d0_Collection_Base
//
// Purpose: Templatized base class for collections.
//          Provides methods which depend on the element type.
//
{
public:
  typedef T d0_value_type;

  virtual ~d0_Collection () {}

  // Insert a new element, in a collection-defined location.
  // (Usually at the end if the collection is ordered.)
  virtual void insert_element (const d0_value_type& elem) = 0;

  // Removes the first element equal to ELEM from the collection.
  virtual void remove_element (const d0_value_type& elem) = 0;

  // Test if the collection contains an element equal to ELEM.
  virtual bool contains_element (const d0_value_type& elem) const = 0;

  // Iterator creation.
  // Only d0_Const_Iterators here; methods for
  // d0_Iterators are provided in d0_Replaceable_Collection.
  virtual d0_Const_Iterator<T> create_iterator () const;
          // same as d0_begin()
  virtual d0_Const_Iterator<T> d0_begin () const = 0;
  virtual d0_Const_Iterator<T> d0_end () const = 0;

  // These abstract methods from d0_Collection_Base are implemented
  // at this point in terms of the above
  //  (element-type dependent) abstract methods.
  void insert_element_ptr (const void* elem);
  void remove_element_ptr (const void* elem);
  bool contains_element_ptr (const void* elem) const;
  d0_Iterator_Base* create_new_iterator () const;
};
```

d0_Replaceable_Collection

```
template <class T>
class d0_Replaceable_Collection
  : public d0_Collection<T>
//
// Purpose: Base class for collections which can have elements
//          replaced
//          through iterators.  Provides methods for creation of
//          d0_Iterator objects.
//
{
public:
  virtual ~d0_Replaceable_Collection () {}

  virtual d0_Iterator<T> create_iterator ();
  virtual d0_Iterator<T> d0_begin () = 0;
  virtual d0_Iterator<T> d0_end () = 0;

  // If these aren't redeclared here, we end up trying to call
  // the above non-const methods, even with a const object.
  virtual d0_Const_Iterator<T> create_iterator () const;
  virtual d0_Const_Iterator<T> d0_begin () const = 0;
  virtual d0_Const_Iterator<T> d0_end () const = 0;
};
```

d0_List_1

```
template <class T>
class d0_List_1
  : virtual public d0_Collection_Base,
    public std::list<T>
//
// Purpose: Intermediate class for list types.
//
{
public:
  // Provide a way of referring to the parent STL class.
  typedef std::list<T> stl_container;

  typedef typename stl_container::value_type value_type;
  typedef typename stl_container::reference reference;
  typedef typename stl_container::const_reference const_reference;
  typedef typename stl_container::size_type size_type;
  typedef typename stl_container::difference_type difference_type;
  typedef typename stl_container::pointer pointer;
  typedef typename stl_container::iterator iterator;
  typedef typename stl_container::const_iterator const_iterator;

  // Standard list constructors.
  d0_List_1 () {}
  d0_List_1 (size_type n, const value_type& value = value_type ());
  d0_List_1 (const value_type* first, const value_type* last);
  d0_List_1 (const stl_container& x);

  // Destructor.
  virtual ~d0_List_1 () {}

  // Assignment.
  d0_List_1<T>& assign_from (const stl_container& x);

  // Standard d0_Collection methods.
  virtual bool is_ordered () const;
  virtual bool allows_duplicates () const;
  virtual int collsize () const;
  virtual int cardinality () const;
```

d0_List_1

```
  virtual bool is_empty () const;
  virtual void remove_all ();

  // Additional ODMG-style methods for accessing and manipulating
  // elements at the head and tail of the list.
  void insert_element_first (const value_type& elem);
  void insert_element_last  (const value_type& elem);
  const_reference retrieve_first_element () const;
  const_reference retrieve_last_element  () const;
  void remove_first_element ();
  void remove_last_element ();

  // Additional ODMG-style methods for concatenating lists.
  d0_List_1<T>& append     (const stl_container& listR);
  d0_List_1<T>& operator+= (const stl_container& listR);
    // equiv. to append

  // Special bulk insertion method.
  virtual void insert_elements (d0om_Collection_Iterator& inserter);

protected:
  // This is a group of operations which are declared as pure virtual
  // functions in d0_Collection and d0_PCollection.  We can't put the
  // implementations of the virtual functions here, since we don't
  // derive from d0_Collection; they have to go in d0_List and d0_PList.
  // However, we would like the d0_PList classes to share as much of this
  // code as possible.  So we implement the actual operations here;
  // the virtual function implementations then just call these routines.

  // Convenient element operations.
  void _insert_element   (const value_type& elem);
  void _remove_element   (const value_type& elem);
  bool _contains_element (const value_type& elem) const;

  // d0_Iterator creation.
  d0_Iterator<value_type>       _d0_begin ();
  d0_Iterator<value_type>       _d0_end ();
  d0_Const_Iterator<value_type> _d0_begin () const;
  d0_Const_Iterator<value_type> _d0_end () const;
};
```

d0_List

```
template <class T>
class d0_List
  : public d0_Replaceable_Collection<T>,
    public d0_List_1<T>
//
// Purpose: Like the list<T> STL class, but with additional ODMG-style
//          operations.
//
{
public:
  // A shorter name for the base list class.
  typedef d0_List_1<T> Base;

  typedef typename Base::stl_container    stl_container;
  typedef typename Base::reference        reference;
  typedef typename Base::value_type       value_type;
  typedef typename Base::const_reference  const_reference;
  typedef typename Base::size_type        size_type;
  typedef typename Base::difference_type  difference_type;
  typedef typename Base::iterator         iterator;
  typedef typename Base::const_iterator   const_iterator;

  // For d0om.
  typedef d0om_d0_Collection_Adapter d0om_collection_adapter;

  typedef d0_Iterator<value_type>       d0_iterator;
  typedef d0_Const_Iterator<value_type> d0_const_iterator;

  // Standard list constructors.
  d0_List () {}
  d0_List (size_type n, const value_type& value = value_type ());
  d0_List (const value_type* first, const value_type* last);
  d0_List (const stl_container& x);

  // Destructor.
  virtual ~d0_List () {}
```

d0_List

```
  // Assignment.
  d0_List<T>& operator= (const stl_container& x);
  d0_List<T>& operator= (const d0_List<T>& x);

  // Convenient element operations.
  virtual void insert_element   (const value_type& elem);
  virtual void remove_element   (const value_type& elem);
  virtual bool contains_element (const value_type& elem) const;

  // Additional ODMG-style methods for concatenating lists.
  d0_List<T> concat    (const stl_container& listR) const;
  d0_List<T> operator+ (const stl_container& listR) const;
    // equiv. to concat

  // D0 Iterator creation.
  virtual d0_Iterator<value_type>       d0_begin ();
  virtual d0_Iterator<value_type>       d0_end ();
  virtual d0_Const_Iterator<value_type> d0_begin () const;
  virtual d0_Const_Iterator<value_type> d0_end () const;
};
```

Other Collections

- Other collection types also define convenience methods and operations appropriate for that type.

- Example: To find the union of two sets:

  - `d0_Set`: `s3 = s1 + s2;`

  - STL `set`:

```
set_union (s1.begin(), s1.end(),
           s2.begin(), s2.end(),
           inserter (s3, s3.begin()));
```

`d0_Iterator` Classes

- Semantics similar to ODMG `d_Iterator` except:
  - Also have `d0_Const_Iterator<T>`. For collections with immutable elements, these are the only iterators which can be created.
  - All derive from a common base class.
- Satisfy the STL `bidirectional_iterator` requirements. But:
  - For each type `T`, there is only one `d0_Iterator<T>` class. STL has a different iterator class for each distinct container.
  - The iterators are tied to the particular collection from which they were created. (Used for `reset()`, `not_done()`, `not_beg()`.)
- Created by `d0_begin()`, `d0_end()`.

d0_Iterator_Base

```
class d0_Iterator_Base
//
// Purpose: Abstract base class for all d0_Iterator and
//          d0_Const_Iterator classes.
//
{
public:
  // Constructor, destructor.
  d0_Iterator_Base () {}
  virtual ~d0_Iterator_Base () {}

  // Reset to the beginning of the associated collection.
  virtual void reset () = 0;

  // Test whether the iterator is at the beginning or end of
  // the collection.
  virtual bool not_done () const = 0;
  virtual bool not_beg () const = 0;

  // Move the iterator forwards or backwards.
  virtual void advance () = 0;
  virtual void backup () = 0;

  // Get back a ‘const void*’ to the referenced element.
  // Signals an ‘iterator exhausted’ exception if the iterator
  // is past the end of the collection.
  // Use of this method outside of the persistence mechanism
  // is discouraged.
  virtual const void* get_ptr () const = 0;
};
```

d0_Iterator

```
template <class T>
class d0_Iterator
  : public d0_Iterator_Base
//
// Purpose: An iterator from which we can get non-const references.
//
{
public:
  // Some type name synonyms.
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;

  // Constructor, destructor.
  d0_Iterator ();
  d0_Iterator (const d0_Iterator<T>& from);
  ~d0_Iterator ();

  // Assignment.
  d0_Iterator<T>& operator= (const d0_Iterator<T>& from);

  // Reset to the beginning of the associated collection.
  void reset ();

  // Test whether the iterator is at the beginning or end
  // of the collection.
  bool not_done () const;
  bool not_beg () const;

  // Move the iterator forwards or backwards.
  void advance ();
  void backup ();
  d0_Iterator<T>& operator++ ();
  d0_Iterator<T>  operator++ (int);
  d0_Iterator<T>& operator-- ();
  d0_Iterator<T>  operator-- (int);
```

```
  // Dereference the iterator.
  // An error will be signalled if the iterator points past the end
  // of the collection.
  reference get_element () const;
  reference operator* () const;
  pointer   operator-> () const;

  // Replace the pointed to element.
  // An error will be signalled if the iterator points past the end
  // of the collection.
  // (An error will also be signaled if the collection is
  //  a mapping type.)
  void replace_element (const value_type& elem);

  // ODMG-style 'next'.
  // If the iterator points past the end of the collection, return false.
  // Otherwise, copy the current element into obj, advance the iterator,
  // and return true.
  bool next (value_type& obj);

  // Comparison.
  // Note that a d0_Iterator<T> cannot be directly compared to a
  // d0_Const_Iterator<T>.  This stems from the fact that not all STL
  // iterator's can be compared with the corresponding const_iterator's.
  friend bool operator== <> (const d0_Iterator<T>& l,
                             const d0_Iterator<T>& r);

  // This is implemented here in terms of get_element().
  const void* get_ptr () const;

  // Make an iterator out of a d0om_STL_Iterator_Base.
  // Used internally by the iterator mechanism.
  d0_Iterator (d0om_STL_Iterator_Base<T>* i);

  friend class d0_Const_Iterator<T>;

private:
  // The actual iterator.
  d0om_STL_Iterator_Base<T>* _it;
};
```

d0om_STL_Iterator_Base

```
template <class T>
class d0om_STL_Iterator_Base
{
//
// Purpose: Base class for STL const_iterator wrappers.
//
public:
  // Some type name synonyms.
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;

  // Constructor, destructor.
  d0om_STL_Iterator_Base () {}
  virtual ~d0om_STL_Iterator_Base () {}

  // Are we at the beginning or end?
  virtual bool not_done () const = 0;
  virtual bool not_beg  () const = 0;

  // Move forwards or backwards.
  virtual void advance () = 0;
  virtual void backup  () = 0;

  // Dereference.
  virtual const_pointer const_ptr () const = 0;
  virtual pointer ptr () const = 0;

  // Make a new instance with equivalent value.
  // (Used when copying d0_Iterator instances.)
  virtual d0om_STL_Iterator_Base<T>* clone () const = 0;
  virtual d0om_STL_Const_Iterator_Base<T>* const_clone () const = 0;
```

d0om_STL_Iterator_Base

```
  // Equality test.
  virtual bool is_eq (const d0om_STL_Iterator_Base<T>& i) const = 0;

  // Reset to the beginning of the collection.
  virtual void reset () = 0;

  // Get back the associated collection.
  virtual const d0_Collection_Base* coll () const = 0;

  // Replace the element to which we're pointing.
  // For a d0om_STL_Iterator, this will signal an error.
  // A functional version is implemented in d0om_STL_Replaceable_Iterator.
  // (This is so we can have non-const d0_Iterator's for mapping types.)
  virtual void replace (const value_type& elem) = 0;
};
```

d0om_STL_Iterator

```
template <class T, class Coll>
class d0om_STL_Iterator
  : public d0om_STL_Iterator_Base<T>
//
// Purpose: STL iterator wrapper.
//
{
public:
  // Make a shorter name for the STL iterator.
  typedef typename Coll::iterator iterator;

  // And the base class.
  typedef d0om_STL_Iterator_Base<T> Base;

  typedef typename Base::value_type      value_type;
  typedef typename Base::pointer         pointer;
  typedef typename Base::const_pointer   const_pointer;
  typedef typename Base::reference       reference;
  typedef typename Base::const_reference const_reference;

  // Construct an STL iterator given the associated collection.
  // The first of these positions the iterator at the beginning of the
  // collection; the second positions the iterator at the end.
  // (The int argument is not actually used.)
  d0om_STL_Iterator (Coll* coll);
  d0om_STL_Iterator (Coll* coll, int);

  // Copy constructor.
  d0om_STL_Iterator (const d0om_STL_Iterator<T, Coll>& i);

  // Destructor.
  virtual ~d0om_STL_Iterator () {}
```

d0om_STL_Iterator

```
  // Implement the d0_Iterator_Base<T> methods here.
  virtual bool not_done () const;
  virtual bool not_beg  () const;
  virtual void advance ();
  virtual void backup  ();
  virtual const_pointer const_ptr () const;
  virtual pointer ptr () const;
  virtual d0om_STL_Iterator_Base<T>* clone () const;
  virtual d0om_STL_Const_Iterator_Base<T>* const_clone () const;
  virtual bool is_eq (const d0om_STL_Iterator_Base<T>& i) const;
  virtual void reset ();
  virtual const d0_Collection_Base* coll () const;
  virtual void replace (const value_type& elem);

  friend class d0om_STL_Const_Iterator<T, Coll>;

private:
  // The STL iterator itself.
  iterator _it;

  // The collection to which it is referring.
  Coll* _coll;
};
```

d0om_STL_Replaceable_Iterator

```
template <class T, class Coll>
class d0om_STL_Replaceable_Iterator
  : public d0om_STL_Iterator<T, Coll>
//
// Purpose: A d0om_STL_Iterator which implements replace().
//
{
public:
  // The base class.
  typedef d0om_STL_Iterator<T, Coll> Base;

  typedef typename Base::value_type      value_type;
  typedef typename Base::pointer         pointer;
  typedef typename Base::const_pointer   const_pointer;
  typedef typename Base::reference       reference;
  typedef typename Base::const_reference const_reference;

  // Construct an STL iterator given the associated collection.
  // The first of these positions the iterator at the beginning of the
  // collection; the second positions the iterator at the end.
  // (The int argument is not actually used.)
  d0om_STL_Replaceable_Iterator (Coll* coll);
  d0om_STL_Replaceable_Iterator (Coll* coll, int);

  // Copy constructor.
  d0om_STL_Replaceable_Iterator
    (const d0om_STL_Replaceable_Iterator<T, Coll>& i);

  // Destructor.
  virtual ~d0om_STL_Replaceable_Iterator () {}

  // Need this to get the type right when copying.
  virtual d0om_STL_Iterator_Base<T>* clone () const;

  // The replace method.
  virtual void replace (const value_type& elem);
};
```

Objects and References

References

- Reference type: `d0_Ref<T>`.
- `T` must derive from `d0_Object`.
- Provides:
  - Deferred reads.
  - Reference counting of objects.
- Usage:

```
d0_Ref<Obj> r;
r->func (r->memvar);
Obj tmp = *r;
Obj* p = r.ptr ();
```

- References point at objects in the same ‘event’ (or constant objects). ‘Longrefs’ planned, but not yet implemented.

Type Conversions

- `d0_Ref<A>` is not convertable to `d0_Ref<B>`, even if `A*` is convertable to `B*`.

- Type conversion macros:
  - Given `d0_Ref<A> r`:
  - `D0_REFCONV((B)) (r)` — Convert `r` to `d0_Ref<B>`. Only allowed if `A*` is convertable to `B*`. Gives a compile-time error otherwise.
  - `D0_CONST_REFCONV((B)) (r)` — Same, except converts to `d0_Ref<const B>`.
  - `d0_REFCAST((B)) (r)` — Convert `r` to `d0_Ref<B>`. Only allowed if `dynamic_cast<B*> (r.ptr())` would work. Gives a run-time error otherwise.
  - `D0_CONST_REFCAST((B)) (r)` — Same, except converts to `d0_Ref<const B>`.
  - `d0_REF_UNSAFECAST((B)) (r)` — Convert `r` to `d0_Ref<B>`. No checking done.
  - `D0_REF_UNSAFECAST((B)) (r)` — Same, except converts to `d0_Ref<const B>`.

- In the future, adding appropriate template conversion functions may make `D0_REFCONV` unneeded.

Reference Counting

- Objects allocated with 'new' are reference-counted by default.

- Example:

```
d0_Ref<foo> a1 = new foo (1);
// foo(1) now has one reference.
{
  d0_Ref<foo> a2 = a1;
  // now it has two references.
  a1 = new foo (2);
  // now foo(1) has one ref. again.
  a2 = a1;
  // all references to foo(1) are
  //  now gone; foo(1) is deleted.
  //  foo(2) now has two references.
}
// a2 goes away.  foo(2) now has
//  one reference.
a1.clear ();
// now foo(2) is deleted.
```

- Can create non-reference counted objects:

```
new (d0_Object::NO_REFCOUNT) foo (2);
```

- Watch out for cycles!

Bare C++ pointers

- No reference counting: must explicitly delete referenced objects.
- No deferred reads. When a structure is read containing C++ pointers, all the referenced objects are also read.
- Cannot have both a C++ pointer and a `d0_Ref` pointing at the same object.
- Target type must derive from `d0_Object`.
- `auto_ptr` treated like bare C++ pointer.
  - Ownership flag is not saved.
  - Unsafe to save a structure with multiple `auto_ptr`'s pointing at the same object.

Implementation

- Design driven by requirement for deferred I/O.
- A `d0_Ref` points at an "indirect pointer": `d0om_Indptr`.
- `d0om_Indptr` is an abstract class, with different concrete classes for different I/O mechanisms.
- `d0_Indptr::makeobj()` should read or otherwise construct the object pointed to and return it.
- A `d0_Object` contains a pointer back to its `d0om_Indptr`. This pointer is null only for objects which are not reference counted. Objects allocated through `new` get a `d0om_Transient_Indptr`.
- The reference count is kept in the `d0om_Indptr`.
- `d0_Ref` also stores a pointer to the object directly, when known.

d0_Object

```
class d0_Object
  : public d0om_Registerable
//
// Purpose: Base class for persistent objects.
//
{
public:
  // Define standard names for pointers to this type.
  typedef d0_Object* d0_pointer;
  typedef const d0_Object* d0_const_pointer;

  // Define flags for operator new.
  enum Refcount_Type {
    NO_REFCOUNT,
    REFCOUNT
  };

  // Allocation and deallocation.
  void* operator new (size_t size, Refcount_Type type = REFCOUNT);
  void* operator new (size_t size, void* p);
  void operator delete (void* p);

  // Additional forms of delete needed for cleaning up on exceptions.
  void operator delete (void* p, Refcount_Type type);
  void operator delete (void* p, void* q);

  // Constructor, destructor.
  d0_Object ();
  d0_Object (const d0_Object& o);
  virtual ~d0_Object ();

  // Assignment.
  d0_Object& operator= (const d0_Object& o);
```

d0_Object

```
// This can be called to mark that the object has changed.
// This is not normally needed, but it's used by d0_Const_Object's,
// and may in the future be useful for objects which were read
// via a database interface.
virtual void mark_modified ();

// The activate and deactivate methods.
virtual void activate ();
virtual void deactivate () const;

// Get the type of this object.  A concrete declaration of this
// is provided in the D0_OBJECT_SETUP macro.
virtual const d0om_Type_Object* d0om_type () const = 0;

// Get this object's indptr.  For internal use.
d0om_Indptr* d0om_indptr () const;

// Get a pointer to the DEST_TYPE subobject of this object.
// Throws a bad_downcast exception if DEST_TYPE is not a base of the
// complete class of this object.  For internal use.
const void* d0om_ptrconv (const d0om_Type_Object* dest_type) const;

// Return the static type of this class.
static d0om_Type_Object* d0om_type_static ();

friend class d0_Ref_Base;
friend class d0om_Indptr;
```

d0_Object

```
private:
  // Change the associated indptr.  Use with caution.
  void d0om_set_indptr (d0om_Indptr* ind);

  // Create a new transient indptr for this object.  Used when creating
  // a reference counted object.
  void d0om_make_transient_indptr ();

  // The associated indptr for this object.  Null if this object is not
  // reference counted.
  d0om_Indptr* _indptr;

  // Set in ‘operator new’ to say that when the constructor runs,
  // it should call d0om_make_transient_indptr().
  static bool _make_transient_indptr_flag;

  // The Type object for d0_Object.
  static d0om_Type_Object* _d0om_type;
};
```

D0_OBJECT_SETUP

```
#define DOOM_DOOM_TYPE_SETUP(name) \
  static const d0om_Type_Object* d0om_type_static \
    (const d0om_Type_Object* typ = 0) \
  { \
    static const d0om_Type_Object* d0om_##name##_type; \
    if (typ) d0om_##name##_type = typ; \
    return d0om_##name##_type; \
  }


#define D0_OBJECT_SETUP(name) \
public: \
  typedef name* d0_pointer; \
  typedef name const* d0_const_pointer; \
  DOOM_DOOM_TYPE_SETUP(name) \
  virtual const d0om_Type_Object* d0om_type () const \
    { return d0om_type_static (); } \
  static name* d0om_make ()
```

d0_Ref_Base

```
class d0_Ref_Base
//
// Purpose: Common base class for references.
//
{
public:
  // Constructors, destructor.
  d0_Ref_Base ();
  d0_Ref_Base (d0om_Indptr* indptr);
  ~d0_Ref_Base ();

  // Clear the reference.
  void clear ();

  // Predicates to check for null references.
  bool is_null() const;
  bool operator!() const;

  // Get the reference back as an object pointer.
  // Will return null if the object is not reference counted and
  // REF_TYPE is not supplied.
  const d0_Object* as_objptr (const d0om_Type_Object* ref_type = 0)
         const;

  // Return the pointer only if the object has been constructed.
  // Return null otherwise.
  // Will not trigger a read operation.
  const void* ptr_only () const;

  // Do these d0_Refs and pointers refer to the same objects?
  inline
  friend bool operator== (const d0_Ref_Base& anyL,
                          const d0_Ref_Base& anyR);

  // For sorting references.
  inline
  friend bool operator< (const d0_Ref_Base& anyL,
                         const d0_Ref_Base& anyR);
```

d0_Ref_Base

```
  // These are made public to avoid a lot of friend declarations,
  // but they are mainly for internal use of the persistence mechanism.
  // Don't use them elsewhere unless you understand the internal
  // workings of references and are prepared to cope with any changes.
  // Assign from an indptr.  For internal use -- no type checking
  // is done.
  d0_Ref_Base& operator= (d0om_Indptr* indptr);

  // Get back the indptr for this reference.
  // Mainly for internal use.
  d0om_Indptr* d0om_ref_indptr () const;

protected:
  // Assignment.  This is protected because it assumes that
  // the references are to the same type.
  d0_Ref_Base& operator= (const d0_Ref_Base& from);

  // Construct a reference directly out of a object pointer
  // and an indptr.  The pointer is assumed to be the correct type.
  d0_Ref_Base (const void* ptr, d0om_Indptr* indptr);

  // Copy constructor.  Assumes that the references are of
  // compatible type.
  // Dummy int arg used to avoid ambiguities with conversions.
  d0_Ref_Base (const d0_Ref_Base& from, int);

  // Copy constructor.  FROM_PTR overrides the _ptr value in FROM.
  // Used for type conversions.
  d0_Ref_Base (const d0_Ref_Base& from, void* from_ptr);

  // Make a reference directly from an object pointer.
  d0_Ref_Base (const d0_Object* o, const d0om_Type_Object* dest_type);

  // Return the reference as a pointer as a DEST_TYPE pointer.
  // Creates the referred to object, if needed.
  const void* ptr (const d0om_Type_Object* dest_type) const;

  // Like ptr(), but signals an exception if the reference is null.
  const void* ptr_check (const d0om_Type_Object* dest_type) const;
```

d0_Ref_Base

```
  // Delete the object to which the reference is pointing.
  // Signals an exception if there are other references to the object.
  // Note that this only handles the case where the object is
  // reference-counted
  // (has an indptr).  It will return true in that case.  If the object
  // is not refcounted, this routine will do nothing and return false.
  // Methods in the derived class must then do the deletion.
  // This is because we need to know the type of the object in order
  // to delete it.
  bool delete_object_base ();

  // Compare to an object referenced by a bare pointer.
  // Actually, we need two representations of the pointer.
  // Once as a d0_Object*, and once as the type corresponding
  // to the derived d0_Ref type.
  bool compare_ptr (const d0_Object* ptr_o, const void* ptr_v) const;

  // Non-inline portion of pointer dereference operation.
  // Called when _ptr is null.
  // If we have an indptr, construct the object.
  const void* ptr1 (const d0om_Type_Object* dest_type) const;


  // Pointer to the object.  Note that the actual type of this pointer
  // will be determined by the derived class.
  mutable const void* _ptr;

private:
  // Increment or decrement the indptr reference counter, provided that
  // the reference is non-null.
  void maybe_incref ();
  void maybe_decref ();

  // The indptr for the object to which we're referring.
  d0om_Indptr* _indptr;
};
```

d0_Ref

```
template <class T>
class d0_Ref
  : public d0_Ref_Base
//
// Purpose: User-visible reference classes.
//
{
public:
  typedef T* pointer;
  typedef T const * const_pointer;
  typedef T element_type;

  // Constructors, destructor.
  d0_Ref () {}
  d0_Ref (const d0_Ref<T>& from);
  d0_Ref (T* from);
  ~d0_Ref () {}

  // Dereferencing.
  T* operator->() const;
  T& operator*()  const;
  T* ptr() const;

  // Return the pointer only if the object has been constructed.
  // Return null otherwise.
  T* ptr_only () const;

  // Get the pointer back as a const void*.
  operator const void* () const;

  // Delete the object to which the reference is pointing.
  // Signals an exception if there are other references to the object.
  void delete_object ();
```

d0_Ref

```
  // This is used internally by the conversion functions, and is not
  // for general use.
  // ??? Need to have template friends before we can make this private.
  d0_Ref (const T* ptr, const d0_Ref_Base& ref);

  friend bool operator== <> (const d0_Ref<T>& a, const T* b);
  friend bool operator== <> (const T* a,         const d0_Ref<T>& b);

private:
  // Non-inline portion of pointer dereference.
  T* ptr1 () const;
};
```

d0_Refcount

```
class d0_Refcount
//
// Purpose: Simple reference-counted object.
//
{
public:
  // Constructor, destructor.
  d0_Refcount ();
  virtual ~d0_Refcount ();

  // Increment and decrement reference count.
  void incref ();
  void decref ();

  // True if calling decref() will delete the object.
  bool decref_will_delete () const;

  // True if incref() has never been called, or if the object is being
  // deleted.
  bool unowned () const;

private:
  // The reference count itself.
  unsigned _refcount;
};
```

d0om_Indptr

```
class d0om_Indptr
  : public d0_Refcount,
    public d0om_Registerable
//
// Purpose: Indirect pointer abstract base class for implementing
//          references and reference counting.
//
{
public:

  // Default constructor.
  d0om_Indptr ();

  // Constructor, specifying the corresponding object.
  d0om_Indptr (d0_Object* o);

  // Destructor.
  virtual ~d0om_Indptr ();

  // Get back a pointer to the corresponding object.
  // If the object does not yet exist, it is created with makeobj().
  d0_Object* ptr ();

  // Instantiate the corresponding object and store a pointer
  // to it in RETPTR.
  // Must be implemented by the method-specific subclasses.
  // nb. The reason we return it via a parameter rather than a function
  // return value is that when making an object is triggered by reading
  // a bare C++ pointer, we must have the pointer initialized properly
  // before we actually start filling in the fields.  Otherwise,
  // an infinite reference loop can result.
  //
  // This implies that the implementation of makeobj should set RETPTR
  // immediately after the object is allocated, before the fields get
  // filled in.
  virtual void makeobj (d0_Object* & retptr) = 0;
```

d0om_Indptr

```
  // If true, then establishing d0_Ref's to this object is prohibited.
  // (Used for objects read in via bare C++ pointers.)
  bool noref () const;

  // Set the noref flag.  The indptr must be unowned for this to work.
  void set_noref ();

  // Clear the object pointer, then delete this object.
  // Used when an object is being deleted, to prevent
  // circular destruction requests.
  void clear_and_delete ();

private:

  // Prohibit copying.
  d0om_Indptr (const d0om_Indptr&);
  d0om_Indptr& operator= (const d0om_Indptr&);

  // Create the object and fill in _obj.
  void setptr ();

  // Pointer to the corresponding object.
  // This is null before the object has been created.
  d0_Object* _obj;

  // If true, then establishing d0_Ref's to this object is prohibited.
  // (Used for objects read in via bare C++ pointers.)
  bool _noref;
};
```

d0om_Transient_Indptr

```
class d0om_Transient_Indptr
  : public d0om_Indptr
//
// Purpose: Indirect pointer implementation for doing refcounting of
//          transient d0_Object instances.
//
{
public:
  // Constructor.  We explicitly specify the corresponding object.
  d0om_Transient_Indptr (d0_Object* o);

  // Destructor.
  ~d0om_Transient_Indptr () {}

  // Dummy makeobj() so that we can allocate instances of this class.
  // This should never be called.
  void makeobj (d0_Object* & retptr);

  // Allocation and deallocation.
  inline void* operator new (size_t size);
  inline void operator delete (void* p);
};
```

Object Registries

- In several contexts, we want to put `d0_Object`'s and `d0om_Indptr`'s on lists.

- Keep list of registries in each object?

  – Too much overhead in objects.

- Search all registries when deleting an object?

  – Too much time overhead.

- Minimize space overhead in objects, and minimize time overhead if objects have not been entered in a registry.

- Compromise:

  – Assign each registry a bit.

  – Keep a mask with each object.

  – When an object is entered in a registry, set the corresponding bit in the mask.

  – When an object is deleted, search only the registries indicated by the bits set in the mask.

d0om_Registerable

```
class d0om_Registerable
//
// Purpose: Base class for objects which can be entered in a
//          d0om_Object_Registry.
//
{
public:
  // Constructor, destructor.
  d0om_Registerable ();
  ~d0om_Registerable ();

  // Called when this object has been entered in some registry.
  void d0om_mark_registered (const d0om_Object_Registry& r) const;

protected:
  // Delete this object from all registries matching REGISTRY_TYPE.
  // A null type means all registries.
  // Called from the dtor if when this object is deleted.
  void d0om_delete_from_registries (const void* registry_type = 0)
     const;

private:
  // Bit masks of registries on which this object has been entered
  // at some time.
  mutable unsigned _registered_mask;
};
```

d0om_Object_Registry

```
class d0om_Object_Registry
//
// Purpose: Holder of pointers to d0om_Registerable's.
//
{
public:
  // Constructor, destructor.
  // REGISTRY_TYPE is a unique pointer to identify the type
  // of the registry.  (We use a pointer rather than an enum
  // so that we don't need to declare them all in one place,
  // but we can still be sure of getting unique values.)
  d0om_Object_Registry (const void* registry_type);
  virtual ~d0om_Object_Registry ();

  // Delete O from this registry, if it exists.
  virtual void delete_obj (const d0om_Registerable* o) = 0;

  // Get back the registry type.
  const void* registry_type () const;

  // Get the bitmask for this registry.
  unsigned registry_mask () const;

  // Delete O with mask MASK from all registries matching
  // REGISTRY_TYPE.  (A null pointer matches everything.)
  static void delete_obj_from_all (const d0om_Registerable* o,
                                   unsigned mask,
                                   const void* registry_type = 0);

  // Dump out the list of registries, for debugging.
  static std::ostream& dump_registry_list (std::ostream& s);

  // The largest bit number we can use.
  enum {
    MAX_BITNUM = (sizeof (unsigned) * 8) - 1
  };
```

d0om_Object_Registry

```
private:
  // The type of this registry.
  const void* _registry_type;

  // The mask for this registry.
  unsigned _registry_mask;

  // Bit number for next registry to be created.
  static int _next_bitnum;
};
```

d0om_Object_Set

```
class d0om_Object_Set
  : public d0om_Object_Registry
//
// Purpose: Maintain a set of d0_Object's, where the objects can be
//          specified either by a d0_Object* or by a d0_Ref.
//          We are careful never to dereference the references.
//          We store these objects in a d0om_Object_Registry, so that
//          they will get removed from the set if they're deleted.
//
{
public:
  d0om_Object_Set ();
  ~d0om_Object_Set () {}

  // Return true if the set is empty.
  bool empty () const;

  // Add an object pointer/reference.
  void add (const d0_Object* o);
  void add (d0_Const_Ref_Any ref);

  // Remove an object pointer/reference.
  void remove (const d0_Object* o);
  void remove (d0_Const_Ref_Any ref);

  // Check to see if an object pointer/reference is in the set.
  bool check (const d0_Object* o) const;
  bool check (d0_Const_Ref_Any ref) const;
  bool check (const d0_Ref_Base& ref,
              const d0om_Type_Object* reftype) const;

  // Dump out the set.
  std::ostream& dump (std::ostream& s, bool mask_addr) const;

  // Implement the d0om_Object_Registry protocol.
  virtual void delete_obj (const d0om_Registerable* o);
```

d0om_Object_Set

```
private:
  //??? Really, d0om_Registerable*.
  //Declare it like this to share instantiations.
  // This set holds pointers to both d0_Object's and d0_Indptr's.
  // If an object has an indptr, the indptr is entered; otherwise,
  // the object itself is entered.
  // The logic below is predicated on the constraint that an indptr
  // may later grow an object, but an object cannot later grow
  // an indptr.
  typedef std::set<d0om_Ptrrep, std::less<d0om_Ptrrep> >
     Object_Set_Type;
  Object_Set_Type _obj_set;

  // Object registry type.
  static int _registry_type;
};
```

Constants

- Objects which derive from `d0_Const_Object` which are logically written only once per file.

- Constant objects kept in DSPACK section 2. When writing an event, if any constant objects have been changed or added, the entire constant record is written before the event record.

- Use `mark_modified` to force objects to be rewritten.

d0_Const_Object

```
class d0_Const_Object
  : public d0_Object
//
// Purpose: Run header objects.
//
{
public:
  // Constructors, destructor.
  d0_Const_Object () {}
  d0_Const_Object (const d0_Const_Object& o);
  virtual ~d0_Const_Object () {}

  // Assignment.
  // Automatically calls mark_modified.
  d0_Const_Object& operator= (const d0_Const_Object& o);

  // Mark that this object has changed.
  virtual void mark_modified ();

  // Return the static type of this class.
  static d0om_Type_Object* d0om_type_static ();

  // Return a pointer which can be used to identify registries
  // containing d0_Const_Objects which need to be cleared by
  // mark_modified.
  static const void* d0om_const_registry_type ();

private:
  // The Type object for d0_Const_Object.
  static d0om_Type_Object* _d0om_type;

  // For d0om_const_registry_type.
  static int _d0om_const_registry_type;
};
```

Output Filters

- Want to be able to control which objects get written.

- An output filter is an instance of a class deriving from `d0_Output_Filter_Base`.

- Should implement

```
bool should_write_object (const d0_Object* o)
     const;
```

- To be able to veto following a `d0_Ref` without dereferencing it, must implement

```
bool should_write_object (const d0_Ref_Base& ref,
                          const d0om_Type_Object*
                                    reftype)
     const;
```

- An output filter is associated with a specific output stream.

- One general-purpose output filter is provided: `d0_Output_Filter`.

d0_Output_Filter_Base

```
class d0_Output_Filter_Base
//
// Purpose: Decide if a given object should be written.
//
{
public:
  // Constructor, destructor.
  d0_Output_Filter_Base () {}
  virtual ~d0_Output_Filter_Base () {}

  // Should this object be written?
  // This method must be provided by the derived class.
  virtual bool should_write_object (const d0_Object* o) const = 0;

  // Should this object be written?
  // Here, the object is identified by a reference base REF
  // and the d0om_Type_Object representing the class to which
  // the reference was declared to point.
  //
  // The default implementation of this translates the reference
  // to an object pointer, and then calls the above method.
  // However, if you want to be able to veto an object reference
  // without also dereferencing it, you'll have to override
  // this.
  virtual bool should_write_object (const d0_Ref_Base& ref,
                                    const d0om_Type_Object* reftype)
    const;
};
```

d0_Output_Filter

```
class d0_Output_Filter
  : public d0_Output_Filter_Base
//
// Purpose: Decide if a given object should be written.
//
//    By default, every object which we can reach by following
//    pointers or references gets written.
//
//    We supply some mechanisms here to better control this.
//    Note that if an object is vetoed, pointers and references
//    from it are not followed.  Thus, vetoing one object
//    implicitly also vetos those objects which are reached
//    only through it.
//
//    We maintain two lists of objects.  The first is the `write'
//    list.  If this list is empty, then all objects pass this
//    test.  Otherwise, an object is written if it is on the list,
//    and is vetoed if it is not on the list.
//    to be written.  The second, `veto', list contains objects
//    which should _not_ be written.
//
//    Note that objects can be added to these lists using either
//    a pointer to the object or a d0_Const_Ref_Any.  In the latter
//    case, a reference which is vetoed by this mechanism will
//    not be dereferenced by writing.
//
//    Also note that the system will automatically delete objects
//    from these lists when the objects are destroyed.
//
//    If an object is neither passed nor vetoed by these tests,
//    we proceed to test on its type.  Note that in order to do this,
//    the object's reference must be dereferenced at this point.
```

d0_Output_Filter

```
//   We again maintain two lists of classes.  The first is the
//   ‘write’ list.  If this list is empty, then all objects
//   pass this test.  Otherwise, an object is written if its
//   type is on the list, nad is vetoed if it is not on the list.
//   This match must be exact; no account is made of
//   inheritance (??? is this what we want?).
//   The second, ‘veto’, list contains classes which should
//   _not_ be written.
//
{
public:
  // Constructor, destructor.
  d0_Output_Filter ();
  virtual ~d0_Output_Filter ();

  // Add or remove objects from the write or veto lists.
  // Objects are given by a const d0_Object*.
  void add_write_object    (const d0_Object* o);
  void remove_write_object (const d0_Object* o);
  void add_veto_object     (const d0_Object* o);
  void remove_veto_object  (const d0_Object* o);

  // Inquire if an object is on the write or the veto list.
  // Objects are given by a const d0_Object*.
  bool object_on_write (const d0_Object* o) const;
  bool object_on_veto  (const d0_Object* o) const;

  // Add or remove objects from the write or veto lists.
  // Objects are given by a const d0_Const_Ref_Any.
  void add_write_object    (d0_Const_Ref_Any ref);
  void remove_write_object (d0_Const_Ref_Any ref);
  void add_veto_object     (d0_Const_Ref_Any ref);
  void remove_veto_object  (d0_Const_Ref_Any ref);

  // Inquire if an object is on the write or the veto list.
  // Objects are given by a const d0_Const_Ref_Any.
  bool object_on_write (d0_Const_Ref_Any ref) const;
  bool object_on_veto  (d0_Const_Ref_Any ref) const;
```

d0_Output_Filter

```
// Add or remove classes from the write or veto lists.
// Classes are represented by their names.
void add_write_class    (const char* name);
void remove_write_class (const char* name);
void add_veto_class     (const char* name);
void remove_veto_class  (const char* name);

// Inquire if a class is on the write or the veto list.
// Classes are represented by their names.
bool class_on_write (const char* name) const;
bool class_on_veto  (const char* name) const;

// Add or remove classes from the write or veto lists.
// Classes are represented by their d0om_Type structures.
void add_write_class    (const d0om_Type_Object* c);
void remove_write_class (const d0om_Type_Object *c);
void add_veto_class     (const d0om_Type_Object* c);
void remove_veto_class  (const d0om_Type_Object* c);

// Inquire if a class is on the write or the veto list.
// Classes are represented by their d0om_Type structures.
bool class_on_write (const d0om_Type_Object* c) const;
bool class_on_veto  (const d0om_Type_Object* c) const;

// Should this type be written?
bool should_write_type (const d0om_Type_Object* otyp) const;
```

d0_Output_Filter

```
  // Implement the d0_Output_Filter_Base protocol.
  // Should this object be written?
  virtual bool should_write_object (const d0_Object* o) const;

  virtual bool should_write_object (const d0_Ref_Base& ref,
                                    const d0om_Type_Object* reftype)
    const;

  // Should the object be written?
  // Here the object is identified by a reference REF.
  bool should_write_object (const d0_Ref<const d0_Object>& ref)
       const;

  // Dump an output filter.  For testing.
  std::ostream& dump (std::ostream& s, bool mask_addr) const;

private:
  // Copying these guys isn't implemented.
  d0_Output_Filter (const d0_Output_Filter&);
  d0_Output_Filter& operator= (const d0_Output_Filter&);

  // Helper to do class name -> d0om_Type conversion.
  static const d0om_Type_Object* lookup_classname (const char* name);

  // Hold the representation of the private data.
  d0_Output_Filterrep* _rep;
};
```

Other Utilities

d0om_Alloc

- A segregated-storage allocator with a STL-like interface.
- Derived from SGI's STL implementation.
- Good for small objects with high turnover, like `d0om_Indptr`'s and collection adapters.

d0om_Alloc

```
template<class T>
class d0om_Alloc
//
// Purpose: STL-like fast allocator for small objects.
//
{
public:
  static T* allocate (size_t n);
  static T* allocate (void);
  static void deallocate (T* p, size_t n);
  static void deallocate (T* p);

  // This avoids a gcc warning.
  friend class d0om_Base_Alloc;

private:
  // Don't make sense for this class.
  d0om_Alloc (const d0om_Alloc<T>&);
  d0om_Alloc<T>& operator= (const d0om_Alloc<T>&);
};
```

Hashmaps

- `d0om_Hashmap` — A simple hash table.
  - Not as full-featured as SGI's `hash_map`, but involves less baggage.

- `d0om_Ptrmap` — A Hashmap designed for mappings between pointers and pointers or pointers and integers.

d0om_Hashmap

```
template <class Key, class Value, class Hasher>
class d0om_Hashmap
//
// Purpose: A simple hash table, mapping KEY to VALUE.
//          HASHER is the hash function object.  It should supply
//          a method init (int tablesize); this gets called during
//          initialization and any time that the table size changes.
//          It should also supply operator(), taking a KEY and
//          returning an integer in the range [0, tablesize).
//
//          Warning about iterators: The iterators defined here can go
//          forwards only.  If the table is modified, any existing
//          iterators will then be invalid for purposes of iterating
//          forward.  They will still be valid, however, for
//          dereferencing
//          an element, or for an argument to erase().  (As long as the
//          element to which the iterator is pointing isn't deleted.)
//
{
public:
  typedef std::pair<const Key, Value> value_type;

  class iterator;

  // Constructor, destructor.
  d0om_Hashmap (const Hasher& hasher = Hasher ());
  ~d0om_Hashmap ();

  // Insert a (key, value) pair.
  bool insert (Key key, Value val, iterator* it = 0);

  // Look up a key.
  bool lookup (Key key, Value& val) const;

  // Delete an entry.
  bool erase (Key key);

  // Remove all entries.
  void clear ();
```

d0om_Hashmap

```
private:
  // The hash table structure is an array of pointers to linked lists
  // of Node structures.

  struct Node {
    Node* next;
    Node** origin;
    std::pair<Key, Value> data;
  };

public:
  // A simple iterator.
  class iterator
  {
  public:
    typedef d0om_Hashmap<Key, Value, Hasher>::value_type value_type;
    typedef value_type& reference;
    typedef value_type* pointer;

    iterator () {}

    reference operator* () const;
    pointer   operator-> () const;

    iterator& operator++ ();
    iterator operator++ (int);
```

```
private:
  // Construct an iterator at the beginning of a table.
  iterator (Node** table);

  // Construct an arbitrary iterator.
  iterator (Node* p, Node** tableptr);

  void advance ();
  Node* _p;
  Node** _tableptr;

  friend class d0om_Hashmap<Key, Value, Hasher>;

public:
  friend bool operator== <> (const iterator& a, const iterator& b);
};

// Iterator creation.
iterator begin ();
iterator end ();

// Erase a node given an iterator.
void erase (iterator i);
```

d0om_Hashmap

```
private:
  typedef d0om_Alloc<Node> node_allocator;

  // The log_2 of the initial size of the table.
  enum { INITIAL_LOG_SIZE = 8 };

  // Increase the size of the hash table.
  void growtable ();

  // The hash function object.
  Hasher _hasher;

  // The alignment parameter, as specified in the ctor.
  int _align;

  // The base-2 log of the current size of the table.
  int _log_tablesize;

  // The current size of the table.
  int _tablesize;

  // The table itself.
  Node** _table;

  // The number of active entries in the table.
  int _nentries;

  // I don't want to deal with writing these...
  d0om_Hashmap (const d0om_Hashmap&);
  d0om_Hashmap& operator= (const d0om_Hashmap&);
};
```

d0om_Ptrmap

```
class d0om_Ptrmap
  : public d0om_Hashmap<d0om_Ptrrep, d0om_Ptrrep, d0om_Ptrrep_Hash>
//
// Purpose: Fast mapping between pointers or integers.
//          Based on the d0om_Hashmap template class.
//
{
public:
  typedef d0om_Hashmap<d0om_Ptrrep, d0om_Ptrrep,
                       d0om_Ptrrep_Hash> Base;

  // Constructor, destructor.
  // ALIGN is the number of low bits in the key which can be ignored
  // when constructing a hash index.
  d0om_Ptrmap (int align = 0);
  ~d0om_Ptrmap () {}

};
```

d0_String

- Derives directly from `string`.

- Works around problems in various implementations of `string`.

- Using `d0_String` instead of `string` (which is really `basic_string<char,char_traits<char>,allocator<char> >`) can substantially shorten mangled names, and thus reduce the size of object files.

Error Reporting

- Presently, most error conditions call `abort()`.

- Will be changed once a standard method of error reporting is settled.

Dictionary

- Decribe layout of the C++ types, providing the information needed for persistence.

- Everything derives from `d0om_Type`.

- Four derived classes:

  - `d0om_Type_Atomic` — Atomic types: Numbers, `bool`, and `string`.

  - `d0om_Type_Collection` — A homogenous collection of another type.

  - `d0om_Type_Reference` — A reference to a `d0_Object`. Includes `d0_Ref`'s, bare pointers, and `auto_ptr`'s.

  - `d0om_Type_Class` — A user-defined class type.

- Classes which derive from `d0_Object` are represented by `d0om_Type_Object`, deriving from `d0om_Type_Class`.

  - For each such class, there's a template class `d0om_Type_Objclass<T>` or `d0om_Type_Abstract_Objclass<T>`. This knows how to do virtual baseclass pointer conversions for `T`, and (for concrete classes) can create a new instance of the described type.

Dictionary Class Creation

- For class types, a static type object is created in the linkage file by the preprocessor.
- For other types, a static `maketype` method is used.
  - Returns an existing type if possible; otherwise, makes a new one.

d0om_Type

```
class d0om_Type
//
// Purpose: Base class for describing persistent object types.
//
{
public:
  // The four kinds of types we can have.
  enum Type_Kind {
    ATOMIC,
    REFERENCE,
    COLLECTION,
    CLASS
  };

  // Constructor, destructor.
  // The constructor takes both the type kind and the printable name
  // of the type.  (Usually, the name is computed by the constructor
  // of the derived class.)
  d0om_Type (Type_Kind kind, const d0_String& name);
  virtual ~d0om_Type ();

  // Return the type's kind and name.
  Type_Kind kind () const;
  const char* name () const;

  // `External' name of the type.
  // Usually the same as name(), but can be overridden to hide some
  // differences between types which shouldn't affect the external
  // representation.
  virtual const char* extname () const;

  // Return true if this type represents a class which derives
  // from d0_Object.
  virtual bool is_object () const;

  // Return true if this type has no persistent data to save.
  virtual bool is_empty () const;
```

d0om_Type

```
// Get the size of this type, in bytes.
// TMPL is a ‘template’ instance of this type.
// (Presently, it’s only needed for d0_Collection types.)
virtual int size (const void* tmpl) const;

// Create a new d0_Object of this type.
// Only defined for types which are actually objects
// (is_object() is true).
// Calling it for other types will abort.
virtual d0_Object* make () const;

// Just after cint initialization, this method gets called
// for all instances which exist at that point.  This gives
// the class types an opportunity to retrieve their definitions
// from cint.
//
// The default implementation is a noop.
virtual void initfields ();

// Generate an error message concerning MEMVAR.
static void error (const d0om_Memvar_Info& memvar,
                   const char* text);
static void error (const d0om_Memvar_Info& memvar,
                   const d0_String& text);

// Returns an iterator which will step through all currently
// defined types.
static d0_Const_PIterator<d0om_Type> type_iterator ();
```

d0om_Type

```
  // Call this once to initialize the type system.
  // Must not be called until after all static d0om_Type_Class
  // objects have been constructed.  Thus, we can't safely do this
  // from a global constructor.
  //
  // Other modules can register functions to be called at
  // initialization time by calling add_initfunc().  These will be
  // called after initializing cint, but before we start calling
  // initfields() on the types.  Thus, we can use this both for the
  // initialization functions for the cint-generated modules, and
  // initialization needed by the output method.
  //
  typedef struct initfunc_struct {
    void (*func) (const char* appname);
    struct initfunc_struct* next;
  } Initfunc;

  static void init (const char* appname);
  static void add_initfunc (Initfunc& p);

  friend class d0om_Cint_Interface;

private:
  // Function type for FILTER_NAME.
  typedef d0_String Name_Filter_t (const d0_String& name);

  // Run the name of this type through the given filter function.
  void filter_name (Name_Filter_t* func);

  // Run the names of all types through the given filter function.
  static void filter_names (Name_Filter_t* func);

  // The kind of this type.
  const Type_Kind _kind;

  // The printable name of this type.
  d0_String _name;
};
```

d0om_Type_Atomic

```
class d0om_Type_Atomic
  : public d0om_Type
//
// Purpose: Describe atomic types.
//
{
public:
  // The possible kinds of atomic types.
  enum Type_Atomic_Kind {
    INT,
    UINT,
    SHORT,
    USHORT,
    LONG,
    ULONG,
    CHAR,
    BOOL,
    FLOAT,
    DOUBLE,
    OCTET,
    STRING,
    ATOMIC_MAX
  };

  // Get back the specific kind of atomic type that this is.
  Type_Atomic_Kind subkind () const;

  // The size of an instance of this type, in characters.
  // TMPL is a ‘template’ instance of this type
  // (not used for atomic types).
  virtual int size (const void* tmpl) const;

  // Return an instance of d0om_Type_Atomic of type SUBKIND.
  // The second arg is not presently used, but would be if we ever
  // wanted to generate an error from here.
  static d0om_Type_Atomic* maketype (Type_Atomic_Kind subkind,
                                     const d0om_Memvar_Info& memvar);
```

d0om_Type_Atomic

```
private:
  // Constructor.  Called by maketype if the requested type does not
  // already exist.
  d0om_Type_Atomic (Type_Atomic_Kind subkind);

  // Construct the printable name corresponding to SUBKIND.
  // Used by the constructor.
  static d0_String makename (Type_Atomic_Kind subkind);


  // The subkind of this instance.
  Type_Atomic_Kind _subkind;

  // Table of d0om_Type_Atomic instances which we've created.
  static d0om_Type_Atomic* _ptrs[ATOMIC_MAX];
};
```

d0om_Type_Collection

```
class d0om_Type_Collection
  : public d0om_Type
//
// Purpose: Describe collection types.
//
{
public:
  // Return the type which this collection contains.
  const d0om_Type* elt () const;

  // The size of an instance of this type, in characters.
  // TMPL is a ‘template’ instance of this type (only needed for
  // d0_Collection types).
  virtual int size (const void* tmpl) const;

  // Return the name of the collection adapter class.
  const char* adapter_name () const;

  // Make a new collection adapter for COLL.
  d0om_Collection_Adapter* make_adapter (void* coll) const;

  // Return an instance of d0om_Type_Collection for a collection
  // containing ELT with full class name NAME.
  // ADAPTER_MAKER is a function which will construct an appropriate
  // collection adapter.  It can be null if the proper collection
  // instance has already been made.
  // ADAPTER_NAME is the name of the adapter class.
  // MEMVAR is be used for generating error messages.
  static d0om_Type_Collection* maketype
    (const d0om_Type* elt,
     const char* name,
     const char* adapter_name,
     d0om_Collection_Adapter_Maker* adapter_maker,
     const d0om_Memvar_Info& memvar);
```

d0om_Type_Collection

```
  // Destructor.
  ~d0om_Type_Collection ();

  // Override this so that all collections with the same value_type
  // will have the same external name.
  virtual const char* extname () const;

private:
  // Constructor.  Called by maketype if the requested type does not
  // already exist.
  d0om_Type_Collection (const d0om_Type* elt,
                        const char* name,
                        const char* adapter_name,
                        d0om_Collection_Adapter_Maker* adapter_maker);

  // Type which this collection contains.
  const d0om_Type* _elt;

  // Name of the collection adapter class.
  d0_String _adapter_name;

  // Function to construct a collection adapter for this instance.
  d0om_Collection_Adapter_Maker* _adapter_maker;

  // Cache the external name.
  d0_String _extname;
};
```

d0om_Type_Reference

```
class d0om_Type_Reference
  : public d0om_Type
//
// Purpose: Describe reference types.
//
{
public:
  // The possible kinds of reference types.
  enum Type_Reference_Kind {
    DOREF,
    REFERENCE,
    POINTER,
    AUTO_PTR
  };

  // Return the specific kind of reference type that this is.
  Type_Reference_Kind subkind () const;

  // Return the type to which this reference is referring.
  const d0om_Type_Object* ref () const;

  // Returns true if this type represents a C++ pointer or reference.
  bool is_cpp_pointer () const;

  // The size of an instance of this reference, in characters.
  // TMPL is a ‘template’ instance of this type
  // (not used for references).
  virtual int size (const void* tmpl) const;

  // Return an instance of d0om_Type_Reference of type SUBKIND
  // referring to REF.  Can return null if there’s an error.
  // MEMVAR is used for generating error messages.
  static d0om_Type_Reference* maketype
           (Type_Reference_Kind subkind,
            const d0om_Type_Object* ref,
            const d0om_Memvar_Info& memvar);

  // Destructor.
  ~d0om_Type_Reference ();
```

d0om_Type_Reference

```
private:
  // Constructor.  Called by maketype if the requested type does not
  // already exist.
  d0om_Type_Reference (Type_Reference_Kind subkind,
                       const d0om_Type_Object* ref);

  // Construct the printable name corresponding to SUBKIND/REF.
  // Used by the constructor.
  static d0_String makename (Type_Reference_Kind subkind,
                             const d0om_Type_Object* ref);

  // The subkind of this instance.
  Type_Reference_Kind _subkind;

  // Type to which this instance refers.
  const d0om_Type_Object* _ref;
};
```

d0om_Type_Class

```
class d0om_Type_Class
  : public d0om_Type
//
// Purpose: Describe class types.
//
{
public:

  // Constructor, destructor.
  // TMPL is a pointer to an instance of the class.  It
  // will be used for computing virtual base offsets.  It can be
  // null if the class is abstract, or if it has no virtual bases.
  // The special value (void*)-1 inhibits warning messages which
  // would ordinarily be caused by a missing template instance.
  d0om_Type_Class (const char* name, const void* tmpl);
  ~d0om_Type_Class ();

  // Retrieve information about persistent member variables:
  // name, type, dimension, offset, and base class flag.  Note that
  // we don't preserve the shape of arrays; 'field_dim' returns the
  // total number of elements in the array.
  int n_fields () const;
  const char* field_name (int i) const;
  const d0om_Type* field_type (int i) const;
  int field_dim (int i) const;
  int field_offs (int i) const;
  bool field_is_base (int i) const;

  // The size of an instance of this class, in characters.
  // TMPL is a 'template' instance of this type (not used for classes).
  virtual int size (const void* tmpl) const;

  // Get back the TMPL parameter passed to the constructor.
  const void* tmpl () const;

  // Initialize the fields for this instance.
  virtual void initfields ();
```

d0om_Type_Class

```
// Mark that initfields should not run on this instance.
// Used to suppress error checking in d0cint in some cases
// where cint screws up.
void skipinit ();

// Returns true if all bases of this class have zero offset.
bool only_simple_inheritance () const;

// Return the offset within this class of the base class BASE.
// Returns -1 if BASE is not a base for this class (or if it's a
// virtual base and the template instance was missing).
int base_offset (const d0om_Type* base) const;

// Return the offset within this class of a d0_Object subobject.
// Returns -1 if this class does not derive from d0_Object.
// Strictly redundant with base_offset() above, but broken out
// to speed up this common case.
int d0_object_offset () const;

// True if this class is empty: neither it nor any base classes have
// any persistent data.
virtual bool is_empty () const;

// Add a new base class to this class's list,
// of type BASE and at offset OFFS.
// IS_VIRTUAL is true if virtual derivation was used,
// and IS_IMMEDIATE if BASE is an immediate base class of
// this class.
void addbase (d0om_Type* base, int offs,
              bool is_virtual, bool is_immediate);

// Add a new field to this class's list.
void addfield (const char* name, d0om_Type* type,
               int offs, int dim);

// Sort the base list entries for fast lookup.
// Should be called after all calls to addbase().
void sortbases ();
```

d0om_Type_Class

```
// Dump a representation of the class to S.
// If MASK is true, replace sizes and offsets with ‘0’.
virtual std::ostream& dump (std::ostream& s, bool mask = false)
  const;

// Look up an instance by class name.
static d0om_Type_Class* find (const char* name);

// Helper to add immediate base info to the field table.
void add_basefields ();

// Return a hash code for this class, which depends on information
// from all persistent fields.
unsigned int hash () const;

// Make known to d0om_Type_Class the function which will do the
// initialization.
// We do this do avoid having d0om_Type_Class depend on
//  d0om_Cint_Interface.
typedef int Class_Initializer (d0om_Type_Class* c);
static void set_initializer (Class_Initializer* initializer);
```

d0om_Type_Class

```
private:
  // Do any needed transformations on the class name.
  // We transform any occurances of ‘string’ to ‘d0_String’.
  d0_String sanitize_name (const char* name);

  // Non-inline portion of base_offset().
  int base_offset_1 (const d0om_Type* base) const;

  // Pointer to the private data.
  struct d0om_Type_Classrep* _rep;

  // Other information which we put here so accesses to them
  // can be inlined.
  // Flag that for this class, all bases have an offset of zero.
  bool _only_simple_inheritance;

  // The offset of a d0_Object subobject in an instance of this class.
  // This is strictly redundant with the info returned by
  // base_offset(); we keep a copy here for speed.
  // This is -1 if this class does not derive from d0_Object.
  int _d0_object_offset;

  // The function which will initialize a d0om_Type_Class.
  static Class_Initializer* _initializer;
};
```

d0om_Type_Classrep

```
// The internal representation for d0om_Type_Class.
struct d0om_Type_Classrep
{
  // Structure to define information for a single member variable.
  struct Field {
    d0_String name;
    const d0om_Type* type;
    int dim;
    int offs;
    bool is_base;
  };


  // Define information about base classes.
  // This is used in the bases table below.
  // Note that this contains information about all bases,
  // not just immediate ones.  A consequence of this is that
  // a base cannot appear twice in a class lattice.
  struct Baseinfo {
    const d0om_Type* type;
    int offs;
    bool is_virtual;
    bool is_immediate;
  };


  // Flag that initfields has been done.
  bool initialized;

  // List of field descriptors.
  std::vector<Field> fields;

  // List of all base classes.
  std::vector<Baseinfo> bases;
```

d0om_Type_Classrep

```
  // List of all base classes, sorted in order by type.
  // Why two copies?  We want one version sorted so that we can quickly
  // find the base record corresponding to a type pointer.  However,
  // since we're sorting on pointers, the ordering is likely to be
  // different on different platforms, causing problems in comparing
  // test results.  (It could also cause a difference in the saved
  // data if multiple inheritance is used.)
  std::vector<Baseinfo> sorted_bases;

  // Size of a class instance.
  int size;

  // Pointer to a static instance of the class.
  // Used as a template for figuring out virtual base offsets.
  const void* tmpl;

  // A mapping of class names to d0om_Type_Class instances.
  // ??? Second template arg should really be d0om_Type_Class*,
  //     but make it a void* for now so we can share the instantiation
  //     with d0om_Collection_Adapter_Registry.
  typedef d0_Map<d0_String, void*> maptype;
  static maptype* classmap;

  // Helper to check validity of a field index.
  void check_index (int i) const;
};
```

d0om_Type_Object

```
class d0om_Type_Object
  : public d0om_Type_Class
//
// Purpose: Common base class for d0om_Type_Objclass<T> types.
//
{
public:
  // Constructor.
  d0om_Type_Object (const char* name, const void* tmpl,
                    bool is_abstract = false);

  // Override this method to return true.
  virtual bool is_object () const;

  // Is this an abstract class?
  bool is_abstract () const;

  // PTR should be a pointer with a declared type the same as the type
  // which this class is describing.  Convert it to a d0_Object pointer.
  virtual const d0_Object* conv_to_objptr (const void* ptr) const;

  // Return true if this type derives from `d0_Const_Object'.
  bool is_const_object () const;

  // Dump a representation of the class to S.
  // If MASK is true, replace sizes and offsets with `0'.
  virtual std::ostream& dump (std::ostream& s, bool mask = false) const;

  // Initialize the fields for this instance.
  virtual void initfields ();

private:
  // True if this type derives from `d0_Const_Object'.
  bool _isconst;

  // True if this class is abstract.
  bool _is_abstract;
};
```

d0om_Type_Abstract_Objclass_Base

```
class d0om_Type_Abstract_Objclass_Base
  : public d0om_Type_Object
//
// Purpose: Base class for d0om_Type_Abstract_Objclass instances.
//
{
public:
  // Constructor.
  d0om_Type_Abstract_Objclass_Base (const char* name);
};
```

d0om_Type_Abstract_Objclass

```
template <class T>
class d0om_Type_Abstract_Objclass
  : public d0om_Type_Abstract_Objclass_Base
//
// Purpose: Describe an abstract d0_Object.
//
{
public:
  // Constructor.
  d0om_Type_Abstract_Objclass (const char* name);

  // Convert to d0_Object pointer.
  virtual const d0_Object* conv_to_objptr (const void* ptr) const;
};
```

d0om_Type_Objclass

```
template <class T>
class d0om_Type_Objclass
  : public d0om_Type_Object
//
// Purpose: Describe a d0_Object class T.
//
{
public:
  // Constructor.
  d0om_Type_Objclass (const char* name, const void* tmpl);

  // Convert to d0_Object pointer.
  virtual const d0_Object* conv_to_objptr (const void* ptr) const;

  // Create a new instance of T, using the default constructor.
  // Really should return T*, but some compilers don't implement
  // covariant returns.
  virtual d0_Object* make () const;
};
```

d0om_Memvar_Info

```
class d0om_Memvar_Info
//
// Purpose: Hold onto a description of a class member variable for use
//          in generating error messages concerning it.
//
{
public:
  // Constructor.  Takes a class name and member variable name.
  d0om_Memvar_Info (const char* classname, const char* varname);

  // Generate an error message TEXT concerning this variable; send
  // it to the stream S.
  std::ostream& error (std::ostream& s, const char* text) const;

private:
  // Saved classname.
  d0_String _classname;

  // Saved variable name.
  d0_String _varname;
};
```

d0om_Cint_Interface

```
class d0om_Cint_Interface
//
// Purpose: Provide an interface to the cint-dependent code.
//
{
public:
  // Query cint and fill in the field info for this class.
  static int initfields (d0om_Type_Class* forclass);

  // cint initialization.
  // If cint gets initialized elsewhere (as in d0cint),
  // cint_initialized() should be called.  This mark the cint
  // initialization has having already been done, so that it will
  // not be repeated by init_cint().
  //
  static void init_cint ();
  static void cint_initialized ();

  // Control whether or not we should warn about incomplete classes.
  static void suppress_incomplete_warnings ();
  static bool incomplete_warnings_suppressed ();

  // Template name hacking.
  static char* extract_template_arg (const char* typ, int n);
  static d0_String maybe_trim_allocator (const d0_String& name_str);
  static void filter_type_names_trim_allocator ();

  // The following two functions are for keeping a map between
  // cint tagnums and d0om_Type pointers.
  static void register_type (int tagnum, d0om_Type* type);
  static void register_type (const char* name, d0om_Type* type);
  static d0om_Type* find_type (int tagnum);
};
```

Initialization Sequence

- `::d0om_init()`
  - `d0om_Cint_Interface::init_cint()` — Initialize `libcint` code.
  - Register `d0om_Cint_Interface::initfields` with `d0om_Type_Class` as the class initializer.
  - `d0om_Type::init()`
    * Call all registered initialization functions.
      · Set up CINT information for each linkage module.
      · Initialize each backend.
    * For each known `d0om_Type` instance `t`, call `t->initfields()`.
      · A no-op for atomic, collection, and reference types.
      · For classes, call the class initializer (`d0om_Cint_Interface::initfields`). Retrieve data member info from CINT and fill in class field info. Other types are created as needed.
  - `d0om_Cint_Interface::` `filter_type_names_trim_allocator()`

`d0cint`

- Based on CINT by M. Goto from HP Japan. (Used in ROOT.)

- Reads C++ headers.

- Writes a C++ *linkage module*. This contains code to define the class structures to CINT. It also contains the static `d0om_Type_Class` definitions for classes, and some other initialization.

- Optionally write a C++ *linkage header*. Including this header will force an external reference to a symbol in the linkage module. This can be used to force the module to be loaded from a library.

- Generates definition information only for classes defined in the top-level header file. `d0cint` should be run on all headers which define classes with persistent data.

- Usage:
  `d0cint` *lnk-module [options] [-ref ref-hdr] input-hdr [-l linkdef-file]*

- *options* can be any option supported by CINT, but only `-D` and `-I` are likely to be useful.

- *linkdef-file* can be used to explicitly specify the classes for which to generate definitions.

`d0cint` Actions

- Scan the input headers and find the classes defined therein. Also notice "`//! +class`" constructions to define template instantiations.
  - *Not* a full C++ parser. In particular, conditional compilation directives are ignored.

- Write the linkdef file. This contains a CINT linkage `#pragma` directive for each class for which we need to generate definitions.

- Construct the CINT argument list from the input arguments.
  - For each option `-I`*path*, add `-I`*path*`/CINT/d0cintinc`.
  - If `cintdefs.hpp` exists in one of the added `d0cintinc` paths, add it as the first source to be processed. This defines `bool` for CINT.
  - Add the linkdef file to the end of the list of sources.
  - If the user specified a linkdef file, append it to the list of sources.

- Call CINT. This produces the CINT portion of the `_lnk.cpp` file.

`d0cint` Actions, continued

- Append `d0om`-specific code to the `_lnk.cpp` file. Includes the static `d0om_Type_Class` definitions and initialization code.

- Construct the `d0om` dictionary from the known classes. This is done only for the purpose of issuing diagnostics at compile time, rather than at run time.

  - Not all checks can be done at this point. For example, `d0_Ref<T>` where `T` is an incomplete type cannot be checked. Full checking must be deferred until runtime where the full definition of `T` must be available.

- If requested, write the linkage header.

Example

```
#include "d0om/d0_Object.hpp"
#include <list>


template <class T>
class A
{
private:
  T _a1;
};


class B
  : public d0_Object
{
public:
  void abs () = 0;
  D0_OBJECT_SETUP (B);

private:
  std::list<int> _b1;
  A<int> _b2;
};


class C
  : public B
{
public:
  D0_OBJECT_SETUP (C);
  void abs ();
};


//! +class A<int>;
```

Linkdef File

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link off all methods;

#pragma link C++ nestedclass;
#pragma link C++ nestedtypedef;

#pragma link off all typedefs;
#pragma link C++ typedef dOom_collection_adapter*;
#pragma link C++ typedef value_type*;
#pragma link C++ typedef element_type*;

#pragma link C++ class B;
#pragma link C++ class C;
#pragma link C++ class A<int>;;

#endif
```

Contents of linkage module

- Header material.

```
/*********************************************************
* x.cc
*********************************************************/
/*#include "x.h.tmp"*/
#ifndef NO_DOOM // for whole file
#ifndef DEFECT_NO_STDLIB_NAMESPACES
namespace std {}
using namespace std;
#endif
#define DOOM_LINKAGE_FILE
#define private public
#define protected public
```

- CINT declarations. (Details omitted.)

- Include source headers:

```
#include "x.hpp"
```

- CINT definitions. (Details omitted.)

Contents of linkage module, continued

- Make static `d0om_Type_Class` definitions.

```
#include "d0om/d0om_Type_Object.hpp"
static d0om_Type_Abstract_Objclass<B > d0om_cls_B ("B");
static d0om_Type_Class d0om_cls_AlEintgR ("A<int>", 0);
static d0om_Type_Objclass<C > d0om_cls_C ("C", 0);
```

- Initialize static types.

```
static void d0om_type_setup ()
{
  B::d0om_type_static (&d0om_cls_B);
  C::d0om_type_static (&d0om_cls_C);
}
```

- Define collection adapters.

```
#include "d0om/d0om_Collection_Adapter_Registry.hpp"
#include "d0om/d0om_STL_Sequence_Adapter.hpp"

static void d0om_collection_setup (){
  d0om_Collection_Adapter_Registry::add (
    "list<int,allocator<int> >",
    d0om_STL_Sequence_Adapter<list<int> >::make);
}
```

Contents of linkage module, continued

- Module initialzation.

```
class d0om_Setup_x {
public:
  d0om_Setup_x ();
};
static void G__cpp_xsetupx(const char*)
{
  G__cpp_setupx ();
  d0om_collection_setup ();
  d0om_type_setup ();
}
d0om_Setup_x::d0om_Setup_x ()
{
  static d0om_Type::Initfunc ii = {G__cpp_xsetupx, 0};
  d0om_Type::add_initfunc (ii);
}
static d0om_Setup_x d0om_setup_x;
```

- Symbols for reference header.

```
// Define hash variables for classes described here.
int d0omhash_B_29b7b66f;
int d0omhash_AlEintgR_2500d98a;
int d0omhash_C_2b689717;
```

Reference Header

```
#ifndef X_REF_HPP
#define X_REF_HPP
#ifndef __CINT__
#ifndef NO_DOOM

// Declare hash variables.
extern int d0omhash_B_29b7b66f;
extern int d0omhash_AlEintgR_2500d98a;
extern int d0omhash_C_2b689717;

static int d0om_externref_X_REF_HPP =
  (&d0omhash_B_29b7b66f - &d0om_externref_X_REF_HPP) +
  (&d0omhash_AlEintgR_2500d98a - &d0om_externref_X_REF_HPP) +
  (&d0omhash_C_2b689717 - &d0om_externref_X_REF_HPP) ;

#endif
#endif
#endif
```