Torrent Systems, Inc.
Five Cambridge Center, Cambridge, MA 02142
tel. 617/354-8484 fax. 617/354-6767

Orchestrate C++ Classes—Sorted By Header File

Header File	*Class*
apt_framework/accessorbase.h	
	APT_AccessorBase
	APT_AccessorTarget
	APT_InputAccessorBase
	APT_InputAccessorInterface
	APT_OutputAccessorBase
	APT_OutputAccessorInterface
apt_framework/adapter.h	
	APT_AdapterBase
	APT_ModifyAdapter
	APT_TransferAdapter
	APT_ViewAdapter
apt_framework/collector.h	
	APT_Collector
apt_framework/composite.h	
	APT_CompositeOperator
apt_framework/config.h	
	APT_Config
	APT_Node
	APT_NodeResource
	APT_NodeSet
apt_framework/cursor.h	
	APT_CursorBase
	APT_InputCursor
	APT_OutputCursor
apt_framework/dataset.h	
	APT_DataSet
apt_framework/fieldsel.h	
	APT_FieldSelector
apt_framework/fifocon.h	
	APT_FifoConnection
apt_framework/gsubproc.h	
	APT_GeneralSubprocessConnection
	APT_GeneralSubprocessOperator
apt_framework/impexp/impexp_function.h	
	APT_GFImportExport
apt_framework/operator.h	
	APT_Operator
apt_framework/partitioner.h	
	APT_Partitioner

Orchestrate C++ Classes—Sorted By Header File *(Continued)*

Header File	*Class*
apt_framework/pipecon.h	
	APT_NullConnection
	APT_PipeConnection
apt_framework/rawfield.h	
	APT_RawField
apt_framework/schema.h	
	APT_Schema
	APT_SchemaAggregate
	APT_SchemaField
	APT_SchemaFieldList
	APT_SchemaLengthSpec
apt_framework/step.h	
	APT_Step
apt_framework/subcursor.h	
	APT_InputSubCursor
	APT_OutputSubCursor
	APT_SubCursorBase
apt_framework/tagaccessor.h	
	APT_InputTagAccessor
	APT_OutputTagAccessor
	APT_ScopeAccessorTarget
	APT_TagAccessor
apt_framework/type/basic/float.h	
	APT_InputAccessorToDFloat
	APT_InputAccessorToSFloat
	APT_OutputAccessorToDFloat
	APT_OutputAccessorToSFloat
apt_framework/type/basic/integer.h	
	APT_InputAccessorToInt16
	APT_InputAccessorToInt32
	APT_InputAccessorToInt64
	APT_InputAccessorToInt8
	APT_InputAccessorToUInt16
	APT_InputAccessorToUInt32
	APT_InputAccessorToUInt64
	APT_InputAccessorToUInt8
	APT_OutputAccessorToInt16
	APT_OutputAccessorToInt32
	APT_OutputAccessorToInt64
	APT_OutputAccessorToInt8
	APT_OutputAccessorToUInt16
	APT_OutputAccessorToUInt32
	APT_OutputAccessorToUInt64
	APT_OutputAccessorToUInt8
apt_framework/type/basic/raw.h	
	APT_InputAccessorToRawField
	APT_OutputAccessorToRawField
	APT_RawFieldDescriptor

Orchestrate C++ Classes—Sorted By Header File *(Continued)*

Header File	*Class*
apt_framework/type/basic/string.h	
	APT_InputAccessorToString
	APT_OutputAccessorToString
	APT_StringDescriptor
apt_framework/type/conversion.h	
	APT_FieldConversion
	APT_FieldConversionRegistry
apt_framework/type/date/date.h	
	APT_DateDescriptor
	APT_InputAccessorToDate
	APT_OutputAccessorToDate
apt_framework/type/decimal/decimal.h	
	APT_DecimalDescriptor
	APT_InputAccessorToDecimal
	APT_OutputAccessorToDecimal
apt_framework/type/descriptor.h	
	APT_FieldTypeDescriptor
	APT_FieldTypeRegistry
apt_framework/type/function.h	
	APT_GenericFunction
	APT_GenericFunctionRegistry
	APT_GFComparison
	APT_GFEquality
	APT_GFPrint
apt_framework/type/protocol.h	
	APT_BaseOffsetFieldProtocol
	APT_EmbeddedFieldProtocol
	APT_FieldProtocol
	APT_PrefixedFieldProtocol
	APT_TraversedFieldProtocol
apt_framework/type/time/time.h	
	APT_TimeDescriptor
	APT_InputAccessorToTime
	APT_OutputAccessorToTime
apt_framework/type/timestamp/timestamp.h	
	APT_TimeStampDescriptor
	APT_InputAccessorToTimeStamp
	APT_OutputAccessorToTimeStamp
apt_framework/utils/fieldlist.h	
	APT_FieldList
apt_util/archive.h	
	APT_Archive
	APT_FileArchive
	APT_MemoryArchive
apt_util/argvcheck.h	
	APT_ArgvProcessor
apt_util/assert.h	
	APT_FatalPath

Torrent Systems, Inc.
Five Cambridge Center, Cambridge, MA 02142
tel. 617/354-8484 fax. 617/354-6767

Orchestrate C++ Classes—Sorted By Header File *(Continued)*

Header File	*Class*
apt_util/date.h	
	APT_Date
apt_util/dbinterface.h	
	APT_DataBaseDriver APT_DataBaseSource APT_DBColumnDescriptor
apt_util/decimal.h	
	APT_Decimal
apt_util/endian.h	
	APT_ByteOrder
apt_util/env_flag.h	
	APT_EnvironmentFlag
apt_util/errind.h	
	APT_Error
apt_util/errlog.h	
	APT_ErrorLog
apt_util/errorconfig.h	
	APT_ErrorConfiguration
apt_util/fast_alloc.h	
	APT_FixedSizeAllocator APT_VariableSizeAllocator
apt_util/fileset.h	
	APT_FileSet
apt_util/identifier.h	
	APT_Identifier
apt_util/keygroup.h	
	APT_KeyGroup
apt_util/locator.h	
	APT_Locator
apt_util/persist.h	
	APT_Persistent
apt_util/proplist.h	
	APT_Property APT_PropertyList
apt_util/random.h	
	APT_RandomNumberGenerator

Orchestrate C++ Classes—Sorted By Header File *(Continued)*

Header File	*Class*
apt_util/rtti.h	
	APT_TypeInfo
apt_util/string.h	
	APT_String APT_StringAccum
apt_util/time.h	
	APT_Time APT_TimeStamp

Orchestrate C++ Macros—Sorted By Header File

Header File	*Macro*
apt_framework/accessorbase.h	
	APT_DECLARE_ACCESSORS() APT_IMPLEMENT_ACCESSORS()
apt_framework/osh_name.h	
	APT_DEFINE_OSH_NAME() APT_REGISTER_OPERATOR()
apt_framework/type/basic/conversions_default.h	
	APT_DECLARE_DEFAULT_CONVERSION() APT_DECLARE_DEFAULT_CONVERSION_WARN()
apt_framework/type/protocol.h	
	APT_OFFSET_OF()
apt_util/archive.h	
	APT_DIRECTIONAL_SERIALIZATION()
apt_util/assert.h	
	APT_ASSERT() APT_DETAIL_FATAL() APT_DETAIL_FATAL_LONG() APT_MSG_ASSERT() APT_USER_REQUIRE() APT_USER_REQUIRE_LONG()
apt_util/condition.h	
	CONST_CAST() REINTERPRET_CAST()
apt_util/errlog.h	
	APT_APPEND_LOG() APT_DUMP_LOG() APT_PREPEND_LOG()
apt_util/exception.h	
	APT_DECLARE_EXCEPTION() APT_IMPLEMENT_EXCEPTION()
apt_util/fast_alloc.h	
	APT_DECLARE_NEW_AND_DELETE() APT_DECLARE_NEW_AND_DELETE_2() APT_IMPLEMENT_NEW_AND_DELETE() APT_IMPLEMENT_NEW_AND_DELETE_2()

Orchestrate C++ Macros—Sorted By Header File *(Continued)*

Header File	*Macro*
apt_util/logmsg.h	
	APT_DETAIL_LOGMSG() APT_DETAIL_LOGMSG_LONG() APT_DETAIL_LOGMSG_VERYLONG()
apt_util/persist.h	
	APT_DECLARE_ABSTRACT_PERSISTENT() APT_DECLARE_PERSISTENT() APT_DIRECTIONAL_POINTER_SERIALIZATION() APT_IMPLEMENT_ABSTRACT_PERSISTENT() APT_IMPLEMENT_ABSTRACT_PERSISTENT_V() APT_IMPLEMENT_NESTED_PERSISTENT() APT_IMPLEMENT_PERSISTENT() APT_IMPLEMENT_PERSISTENT_V()
apt_util/rtti.h	
	APT_DECLARE_RTTI() APT_DYNAMIC_TYPE() APT_IMPLEMENT_RTTI_BASE() APT_IMPLEMENT_RTTI_BEGIN() APT_IMPLEMENT_RTTI_END() APT_IMPLEMENT_RTTI_NOBASE() APT_IMPLEMENT_RTTI_ONEBASE() APT_NAME_FROM_TYPE() APT_PTR_CAST() APT_STATIC_TYPE() APT_TYPE_INFO()

```
// -*-Mode: C++-*-
// Copyright (c) 1996, 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ACCESSORBASE_H
#define APT_ACCESSORBASE_H

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_FIELDSEL_H
#include <apt_framework/fieldsel.h>
#endif

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_PROTOCOL_H
#include <apt_framework/type/protocol.h>
#endif

class APT_AccessorBase;
class APT_OutputAccessorBase;
class APT_Archive;
class APT_InputAccessorInterface;
class APT_OutputAccessorInterface;
class APT_InputTagAccessor;
class APT_OutputTagAccessor;
class APT_InputSubCursor;
class APT_OutputSubCursor;
class APT_InputInterface;
class APT_OutputInterface;
class APT_InterfaceRep;
class APT_Schema;
class APT_FieldSelector;
class APT_DMAccessor;

/* You can define a preprocessor symbol APT_ACCESSOR_NOCHECK (at the
   top of your compilation unit, before any includes).  If this symbol
   is defined before this header file is included, then all validity
   checks on accessor dereferencing are suppressed.  The usual caveats
   apply.

   By default, the APT_ACCESSOR_NOCHECK symbol is not defined, so by
   default accessor dereferencing checks are enabled.
*/
```

```
class APT_AccessorTarget
/* Synthetic base class that we mix into the APT_Interface internals, so
   that we can hide most of the APT_Interface implementation and yet have
   these crucial accessor-related functions be inline.

   Torrent note: APT_InterfaceRep::Field is the only class that may have
   APT_AccessorTarget as a base class.
*/
{
protected:
  friend class APT_AccessorBase;
  friend class APT_OutputAccessorBase;
  friend class APT_InterfaceRep;

  APT_AccessorTarget();
  ~APT_AccessorTarget();

  APT_UInt32 vecMode_f;         /* 0: var-length; positive: fixed-length */

  const APT_FieldTypeDescriptor* type_f;
  bool nullable_f;

  const bool* presentp_;        /* points to flag that will be false if
                                   this field is within a scope vector
                                   whose number of occurrences is
                                   zero; the flag updated to point into this
                                   field's concrete scope. */
  const bool* tagActivep_;      /* points to flag updated if this
                                   field's concrete counterpart is an
                                   arm of a tagged; always true otherwise */

  /* pointer to value (or vector of values) of field's interface
     value type. Refers either to our Tvec_, or to same vector as our
     concrete/cascadeDown_f component's vecp_/vecOffset_ */
  char* *vecp_;
  APT_UInt32 vecOffset_;

  /* pointer to null vector for this field.  Refers either to our nullVec_,
     or to same null vector as our concrete/cascadeDown_f component */
  APT_UInt8* *nullVecp_;        // 0 if this component non-nullable
  APT_UInt32* nullableIndex_;
  int nullableIndexOffset_;

  APT_UInt32* activeLenp_;            /* for var-len vector, elements of value
                                   vector actually active for this record;
                                   if non-null, always points to concrete
                                   field's activeLen_ */

  bool hasAccessor_;            /* true if this field has (or has ever had) an
                                   accessor */
```

```
APT_InterfaceRep* intfRep_f;

/* functions for APT_AccessorBase */

bool isUsable() const { return *presentp_ && *tagActivep_; }

bool isVector() const { return vecMode_f != 1; }
bool isFixedLengthVector() const { return vecMode_f > 1; }

APT_UInt32 vectorLength() const { if (vecMode_f) return vecMode_f;
                                  else
                                  { checkUsable();
                                    return *activeLenp_;
                                  }
                                }
void setVectorLength_(APT_UInt32);

APT_FieldSelector interfaceField() const;
APT_FieldSelector concreteField() const;

void* base(APT_UInt32 i) { checkUsable();
                           checkBounds(i);
                           if (isNullable())
                           {
                             /* returning non-const pointer; we presume that a
                                write will occur and thus clear the null flag */
                             clearIsNull_(i);
                           }
                           return base_(i);
                         }
const void* base_const(APT_UInt32 i) const { checkUsable();
                                             checkBounds(i);
                                             if (isNullable() && isNull_(i))
                                               nullValue();
                                             return base_(i);
                                           }

bool isNull(APT_UInt32 i) const { checkUsable();
                                  checkBounds(i);
                                  if (!isNullable()) return false;
                                  return isNull_(i);
                                }
void setIsNull(APT_UInt32 i) { checkUsable();
                               checkBounds(i);
                               if (!isNullable()) notNullable();
                               setIsNull_(i);
                             }
void clearIsNull(APT_UInt32 i) { checkUsable();
                                 checkBounds(i);
                                 if (isNullable()) clearIsNull_(i);
                               }
```

```
  void boundAccessor();

private:
  // prohibit copy/assign
  APT_AccessorTarget(const APT_AccessorTarget&);
  APT_AccessorTarget& operator= (const APT_AccessorTarget&);

  void checkUsable() const
  {
#ifndef APT_ACCESSOR_NOCHECK
    if (!isUsable()) notUsable();
#endif
  }

  // requires: isUsable()
  void checkBounds(APT_UInt32 i) const
  {
#ifndef APT_ACCESSOR_NOCHECK
    if (i >= vectorLength_()) badBounds(i);
#endif
  }

  void notUsable() const;
  void badBounds(APT_UInt32 i) const;
  void nullValue() const;
  void notNullable() const;

  bool isNull_(APT_UInt32 i) const
  { int bit = *nullableIndex_ + nullableIndexOffset_ + i;
    return (bool) (((*nullVecp_)[bit>>3] >> (bit&7)) & 1);
  }

  void setIsNull_(APT_UInt32 i)
  { int bit = *nullableIndex_ + nullableIndexOffset_ + i;
    (*nullVecp_)[bit>>3] |= 1 << (bit&7);
    type_f->clearValueType(base_(i));
  }

  void clearIsNull_(APT_UInt32 i)
  { int bit = *nullableIndex_ + nullableIndexOffset_ + i;
    (*nullVecp_)[bit>>3] &= ~(1 << (bit&7));
  }

  APT_UInt32 vectorLength_() const
  { if (vecMode_f) return vecMode_f;
    else return *activeLenp_; }

public:
  APT_UInt32 vecMode() const { return vecMode_f; }
  const APT_FieldTypeDescriptor* type() const { return type_f; }
  bool isNullable() const { return nullable_f; }
```

```
  // no validity checks
  void* base_() { return *vecp_ + vecOffset_; }
  const void* base_() const { return *vecp_ + vecOffset_; }

  void* base_(APT_UInt32 i) { char* b = *vecp_ + vecOffset_;
                              if (i) b += i * type_f->sizeofT();
                              return b;
                            }
  const void* base_(APT_UInt32 i) const { const char* b = *vecp_ + vecOffset_;
                                          if (i) b += i * type_f->sizeofT();
                                          return b;
                                        }
};


class APT_AccessorBase
/* Abstract base class for field data accessor classes.

   APT operators and partitioners use field accessors to effect
   type-safe and efficient read and write access to record data at
   runtime.  In addition, code that directly accesses datasets also
   use accessors to read and write record data.
*/
{
  APT_DECLARE_RTTI(APT_AccessorBase); // DEPRECATED

public:
  enum Type
  {
    /* list of the Torrent-provided field types */
    eInt8=0, eUInt8, eInt16, eUInt16, eInt32, eUInt32, eInt64, eUInt64,
    eSFloat, eDFloat, eString, eRaw,
    eDate, eTime, eTimeStamp, eDecimal,
    eOther=-1                   // not a Torrent-provided field type
  };

  virtual ~APT_AccessorBase();

  bool hasType() const { return type_ != 0; }
  /*
     effect    Tells if this accessor has type information.
               A default-constructed accessor of base type
               (APT_InputAccessorBase or APT_OutputAccessorBase) is
               untyped until it is setup (see
               APT_{Input,Output}AccessorInterface::setupAccessor()).
  */

  Type type() const;
  /*
     effect    Returns the v1.1 field type of this accessor.
     requires  hasType() must be true.
     DEPRECATED
```

```
*/

const APT_FieldTypeDescriptor* typeDescriptor() const;
/*
   effect    Returns the type of this accessor.
   note      The returned pointer refers to the descriptor owned by
             this accessor.
   requires  hasType() must be true.
*/

bool isSetup() const { return target_ != 0; }
/*
   effect    Tells if this accessor has been bound to an interface
             component.
*/

bool isUsable() const { return target_ && target_->isUsable(); }
/*
   effect    Tells if this accessor is currently usable to access
             data (can be dereferenced).
             False if:
             - isSetup() is false, or
             - this is an input accessor and
               APT_InputCursor::getRecord() has never been called
               for the dataset to which this accessor is bound, or
             - getRecord() has returned false, or
             - this is an input accessor and it is bound to a field
               within a tagged aggregate that is not "active".
*/

bool operator! () const { return !isUsable(); }
/*
   effect    Same as !isUsable().  That is, "if (!accessor) { ... }"
             is the same as "if (!accessor.isUsable()) { ... }"
*/

bool isVector() const { checkSetup();
                        return target_->isVector(); }
/*
   effect    Tells whether this field accessor is bound to a field
             that is a single value (false) or a vector (true).
   requires  isSetup() is true.
*/

bool isFixedLengthVector() const { checkSetup();
                                   return target_->isFixedLengthVector(); }
/*
   effect    Tells whether this field accessor is bound to a field
             that a fixed-length vector.
   requires  isSetup() is true.
*/
```

```
APT_UInt32 vectorLength() const { checkSetup();
                                  return target_->vectorLength(); }
/*
   effect    Gives the vector length of the field to which this
             accessor is bound.
             Returns 1 if the field is not a vector.
   requires  isUsable() or isFixedLengthVector() is true.
*/

bool isNullable() const { checkSetup();
                          return target_->isNullable(); }
/*
   effect    Indicates if this accessor is bound to a nullable
             interface field.
   requires  isSetup() is true.
*/

bool isNull() const { checkSetup();
                      return target_->isNull(0); }
bool isNullAt(APT_UInt32 i) const { checkSetup();
                                    return target_->isNull(i); }
/*
   effect    Tells if the interface field to which this accessor
             is bound currently has a null value.
   note      If isNullable() is false, then isNull() always returns
             false.
   requires  isUsable is true.
*/

APT_FieldSelector interfaceField() const { checkSetup();
                                           return target_->interfaceField();}
/*
   effect    Returns the name of the interface component to which this
             accessor is bound.
   requires  isSetup() is true.
*/
APT_FieldSelector concreteField() const { checkSetup();
                                          return target_->concreteField(); }
/*
   effect    Returns the name of the record component to which this
             accessor is bound.
   requires  isSetup() is true.
*/

friend APT_Archive& operator|| (APT_Archive& ar, APT_AccessorBase&)
{ return ar; }
friend APT_Archive& operator<< (APT_Archive& ar, const APT_AccessorBase&)
{ return ar; }
friend APT_Archive& operator>> (APT_Archive& ar, APT_AccessorBase&)
{ return ar; }
/*
   effect    Accessor serialization is supported so that operators'
```

```
              accessor data members can be serialized along with
              other operator data members.
              However, accessor serialization is a nop: the "setup"
              state of an accessor is not serialized.
  */

protected:
  APT_AccessorBase();
  /*
     effect    Initializes this field accessor as "untyped"
               isSetup() is initially false; isUsable() is initially
               false.
  */

  void initType(const char* schemaTypeName);

  void* base(APT_UInt32 i)
  { checkSetup();
    return target_->base(i);
  }
  const void* base(APT_UInt32 i) const
  { checkSetup();
    return target_->base_const(i);
  }

public:
  // for impexp copy combining; archaic but no need to change it
  enum Prot { eEmbedded=0, eBaseOffset };
  bool sameBasePointer(const APT_AccessorBase& other,
                       Prot myProt, APT_UInt32 myOffsetOfBasep,
                       Prot otherProt, APT_UInt32 otherOffsetOfBasep) const;
  APT_UInt32 relativeOffset(const APT_AccessorBase& other,
                        Prot myProt, APT_UInt32 myOffsetOfOffset,
                        Prot otherProt, APT_UInt32 otherOffsetOfOffset) const;

  void* buffer(Prot prot, APT_UInt32 offsetOfBasep, APT_UInt32 offsetOfOffset)
  // requires: accessor must be setup.
  { char* valp = (char*) target_->base_();
    if (prot == eEmbedded) return valp;
    else
    { char** basep = *((char***) (valp + offsetOfBasep));
      APT_UInt32 offset = *((APT_UInt32*) (valp + offsetOfOffset));
      return *basep + offset;
    }
  }

  // for transfer copy-combining optimization-- more modern

  void** basep(APT_BaseOffsetFieldProtocol* bp) const;
  APT_UInt32 offset(APT_BaseOffsetFieldProtocol* bp) const;
  // requires: accessor must be setup.
```

```
  APT_UInt8* nullVec() const;   // requires: nullable
  APT_UInt32 nullOffset() const;        // requires: nullable
  // requires: accessor must be setup.

  // Torrent only!
  // all require: accessor has been setup on a scalar
  void* base_() { return target_->base_(); }
  void* base_(APT_UInt32 i) { return target_->base_(i); } // vector??
  void* buffer(APT_BaseOffsetFieldProtocol* bop)
  { char* valp = (char*) target_->base_();
    if (bop)
    { char** basep = *((char***) (valp + bop->offsetOfBasep()));
      APT_UInt32 offset = *((APT_UInt32*) (valp + bop->offsetOfOffset()));
      return *basep + offset;
    }
    else // embedded
      return valp;
  }
  const APT_AccessorTarget* target() const { return target_; }
  APT_AccessorTarget* target() { return target_; }

  void checkSetup() const
  {
#ifndef APT_ACCESSOR_NOCHECK
    if (!isSetup()) notSetup();
#endif
  }

  void checkUsable() const
  {
#ifndef APT_ACCESSOR_NOCHECK
    if (!isUsable()) notUsable();
#endif
  }

protected:
  friend class APT_InterfaceRep;
  friend class APT_DMAccessor;

  const APT_FieldTypeDescriptor* type_;
  APT_AccessorTarget* target_;

  void notSetup() const;
  void notUsable() const;

  void setVectorLength_(APT_UInt32 i) { target_->setVectorLength_(i); }

private:
  // prohibit copying
  APT_AccessorBase(const APT_AccessorBase&);
  APT_AccessorBase& operator= (const APT_AccessorBase&);
};
```

```
class APT_InputAccessorBase : public APT_AccessorBase
{
  APT_DECLARE_RTTI(APT_InputAccessorBase); // DEPRECATED

public:
  APT_InputAccessorBase();
  /*
     effect    Constructs a "untyped" input accessor, which can be
               setup to access an input field of any type.
  */
  APT_InputAccessorBase(const APT_FieldSelector& component,
                        APT_InputAccessorInterface* cur);
  /*
     effect    Initializes this accessor object to the indicated schema
               component accessible using the given cursor object.
     requires  cur non-null
               cur->isSetup() is true.
               component must identify a component of the cursor's
               schema().
               component must not be subscripted.
               The indicated component must not have been setup with
               an accessor already.
  */

  // copy/assign prohibited in base class

  static APT_InputAccessorBase* allocAccessor(APT_AccessorBase::Type);
  /*
     effect    Allocates an input accessor of the indicated type.
     requires  The type must not be eOther
     DEPRECATED
  */

  const void* value() const { return base(0); }
  const void* valueAt(APT_UInt32 i) const { return base(i); }
  /*
     effect    Dereferences this accessor, providing read-only access
               to the field value; the returned pointer is untyped
               because this accessor is untyped.
     note      The returned pointer is not guaranteed to be valid
               after a record advance.
     requires  isUsable() is true
               isNull() is false
  */

protected:
  void setup(const APT_FieldSelector& component,
             APT_InputAccessorInterface* cur);
};
```

```
class APT_OutputAccessorBase : public APT_AccessorBase
{
  APT_DECLARE_RTTI(APT_OutputAccessorBase); // DEPRECATED

public:
  APT_OutputAccessorBase();
  /*
     effect    Constructs a "untyped" output accessor, which can be
               setup to access an output field (or scratch field) of
               any type.
  */
  APT_OutputAccessorBase(const APT_FieldSelector& component,
                         APT_OutputAccessorInterface* cur);
  /*
     effect    Initializes this accessor object to the indicated schema
               component accessible using the given cursor object.
     requires  cur non-null
               cur->isSetup() is true.
               component must identify a component of the cursor's
               schema().
               component must not be subscripted.
               The indicated component must not have been setup with
               an accessor already.
  */

  // copy/assign prohibited in base class

  static APT_OutputAccessorBase* allocAccessor(APT_AccessorBase::Type);
  /*
     effect    Allocates an output accessor of the indicated type.
     requires  The type must not be eOther
     DEPRECATED
  */

  void setVectorLength(APT_UInt32);
  /*
     effect    Sets the length of the field vector to which this
               output accessor is bound.
               If the length is reduced, field elements at the top of
               the vector are discarded.  If the length is increased,
               the new field elements at the top of the vector are
               defaulted.
     requires  0 <= len <= 0x7fffffff
               isVector() is true.
               isFixedLengthVector() is false.
               isUsable() is true.
  */

  void* value() { return base(0); }
  void* valueAt(APT_UInt32 i) { return base(i); }
  /*
     effect    Dereferences this accessor, providing access to the
```

```
              field value; the returned pointer is untyped because
              This accessor is untyped.
     note     If isNullable() is true, then an implicit clearIsNull()
              operation is performed by this function.
              The returned pointer is not guaranteed to be valid
              after a record advance.
     requires isUsable() is true
  */

  void setIsNull() { checkSetup();
                     target_->setIsNull(0); }
  void setIsNullAt(APT_UInt32 i) { checkSetup();
                                   target_->setIsNull(i); }
  /*
     effect   Sets the null indicator for this accessor's field.
     note     Whenever a putRecord() is performed, all nullable
              fields' null indicators are set; obtaining write access
              to the field (via value()) clears the null indicator.
     requires isUsable() is true.
              isNullable() is true.
  */

  void clearIsNull() { checkSetup();
                       target_->clearIsNull(0); }
  void clearIsNullAt(APT_UInt32 i) { checkSetup();
                                     target_->clearIsNull(i); }
  /*
     effect   Clears the null indicator for this accessor's field.
     note     Whenever a putRecord() is performed, all nullable
              fields' null indicators are set; obtaining write access
              to the field (via value()) clears the null indicator.
              If isNullable() is false, then this function is a nop.
     requires isUsable() is true.
  */


protected:
  void setup(const APT_FieldSelector& component,
             APT_OutputAccessorInterface* cur);
};


/* Template macros for generating new accessor classes for accessing
   field types */

/* VT: value type name
   ST: schema type name
   AT: accessor type name
   Accessor classes will be of the form:
     APT_InputAccessorToAT, APT_OutputAccessorToAT
*/
#define APT_DECLARE_ACCESSORS(VT, ST, AT)                              \
```

```
class APT_InputAccessorTo ## AT : public APT_InputAccessorBase          \
{                                                                       \
  APT_DECLARE_RTTI(APT_InputAccessorTo ## AT); /* DEPRECATED */         \
public:                                                                 \
  APT_InputAccessorTo ## AT();                                          \
  APT_InputAccessorTo ## AT(const APT_FieldSelector& component,         \
                            APT_InputAccessorInterface* cur);           \
  /* requires: The component's type (as declared in the cursor's       \
       schema()) must exactly match the accessor's datatype. */         \
                                                                        \
  const VT& value() const { return *base(0); }                          \
  const VT& operator* () const { return value(); }                      \
  const VT* operator-> () const { return base(0); }                     \
                                                                        \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }            \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }              \
  /* requires  0 <= i < vectorLength() */                               \
                                                                        \
private:                                                                \
  const VT* base(APT_UInt32 i) const                                    \
    { return (const VT*) APT_InputAccessorBase::valueAt(i); }           \
};                                                                      \
class APT_OutputAccessorTo ## AT : public APT_OutputAccessorBase        \
{                                                                       \
  APT_DECLARE_RTTI(APT_OutputAccessorTo ## AT); /* DEPRECATED */        \
public:                                                                 \
  APT_OutputAccessorTo ## AT();                                         \
  APT_OutputAccessorTo ## AT(const APT_FieldSelector& component,        \
                             APT_OutputAccessorInterface* cur);         \
  /* requires: The component's type (as declared in the cursor's       \
       schema()) must exactly match the accessor's datatype. */         \
                                                                        \
  const VT& value() const { return *base(0); }                          \
  VT& writableValue() { return *base(0); }                              \
  void setValue(const VT& val) { *base(0) = val; }                      \
  const VT& operator* () const { return value(); }                      \
  VT& operator* () { return *base(0); }                                 \
  const VT* operator-> () const { return base(0); }                     \
  VT* operator-> () { return base(0); }                                 \
                                                                        \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }            \
  VT& writableValueAt(APT_UInt32 i) { return *base(i); }                        \
  void setValueAt(APT_UInt32 i, const VT& val) { *base(i) = val; }              \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }              \
  VT& operator[] (APT_UInt32 i) { return *base(i); }                    \
  /* requires  0 <= i < vectorLength() */                               \
                                                                        \
private:                                                                \
  const VT* base(APT_UInt32 i) const                                    \
    { return (const VT*) APT_AccessorBase::base(i); }                   \
  VT* base(APT_UInt32 i) { return (VT*) APT_OutputAccessorBase::valueAt(i); }\
}
```

```
#define APT_IMPLEMENT_ACCESSORS(VT, ST, AT)                              \
APT_InputAccessorTo ## AT::APT_InputAccessorTo ## AT()                   \
  : APT_InputAccessorBase()                                              \
{                                                                        \
  initType(#ST);                                                         \
}                                                                        \
APT_InputAccessorTo ## AT::APT_InputAccessorTo ## AT(                    \
                                  const APT_FieldSelector& component, \
                                  APT_InputAccessorInterface* cur)   \
  : APT_InputAccessorBase()                                              \
{                                                                        \
  initType(#ST);                                                         \
  setup(component, cur);                                                 \
}                                                                        \
APT_IMPLEMENT_RTTI_ONEBASE(APT_InputAccessorTo ## AT,                    \
                           APT_InputAccessorBase);                       \
                                                                         \
APT_OutputAccessorTo ## AT::APT_OutputAccessorTo ## AT()                 \
  : APT_OutputAccessorBase()                                             \
{                                                                        \
  initType(#ST);                                                         \
}                                                                        \
APT_OutputAccessorTo ## AT::APT_OutputAccessorTo ## AT(                  \
                                  const APT_FieldSelector& component, \
                                  APT_OutputAccessorInterface* cur)  \
  : APT_OutputAccessorBase()                                             \
{                                                                        \
  initType(#ST);                                                         \
  setup(component, cur);                                                 \
}                                                                        \
APT_IMPLEMENT_RTTI_ONEBASE(APT_OutputAccessorTo ## AT,                   \
                           APT_OutputAccessorBase)


/* uniform interfaces for setting up accessors (papers over the
   previous cursor/interface dichotomy when setting up accessors) */

class APT_InputAccessorInterface
{
public:
  APT_InputAccessorInterface();
  APT_InputAccessorInterface(APT_InputInterface*);

  void setupAccessor(const APT_FieldSelector& component,
                     APT_InputAccessorBase* acc);
  void setupTagAccessor(const APT_FieldSelector& component,
                        APT_InputTagAccessor* acc);
  void setupSubCursor(const APT_FieldSelector& component,
                      APT_InputSubCursor* sub);
  /*
     requires  acc non-null
```

```
              component must not be subscripted.
              The accessor object must not have been setup already.
              isSetup() is true.
  */

  bool isSetup() const { return intf_ != 0; }
  APT_Schema schema() const;

protected:
  void setup(APT_InputInterface*);

private:
  APT_InputInterface* intf_;
};

class APT_OutputAccessorInterface
{
public:
  APT_OutputAccessorInterface();
  APT_OutputAccessorInterface(APT_OutputInterface*);

  void setupAccessor(const APT_FieldSelector& component,
                     APT_OutputAccessorBase* acc);
  void setupTagAccessor(const APT_FieldSelector& component,
                        APT_OutputTagAccessor* acc);
  void setupSubCursor(const APT_FieldSelector& component,
                      APT_OutputSubCursor* sub);
  /*
     requires  acc non-null
               component must not be subscripted.
               The accessor object must not have been setup already.
               isSetup() is true.
  */

  bool isSetup() const { return intf_ != 0; }
  APT_Schema schema() const;

protected:
  void setup(APT_OutputInterface*);

private:
  APT_OutputInterface* intf_;
};


#endif // APT_ACCESSORBASE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ADAPTER_H
#define APT_ADAPTER_H

#ifndef APT_FIELDSEL_H
#include <apt_framework/fieldsel.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_SCHEMA_H
#include <apt_framework/schema.h>
#endif

class APT_FieldTypeDescriptor;
class APT_FieldConversion;
class APT_AdapterRep;
class APT_ParseError;
class APT_Conversion;
class APT_ViewAdapter;
class APT_ModifyAdapter;
class APT_TransferAdapter;
class APT_Lexer;
class APT_ErrorLog;
class APT_Schema;

class APT_StringMap
{
public:
  APT_StringMap(const APT_StringMap&);
  APT_StringMap& operator= (const APT_StringMap&);
  ~APT_StringMap();

  enum CaseSensitivity { eUseCase, eIgnoreCase };
  APT_StringMap(CaseSensitivity cs=eIgnoreCase);
  /*
     effect    Initially empty.  Argument specifies whether all
```

```
              string comparisons are case dependent or case
              independent.
   note       Regardless of the case sensitivity, the case of
              strings is preserved.
*/
void setCaseSensitivity(CaseSensitivity);

APT_StringMap(const char*, APT_ParseError* err=0);
APT_StringMap(const APT_String&, APT_ParseError* err=0);
/*
   effect     Parses the input, expecting a string map description
              (*not* starting with the token 'stringmap').
              If the parse is successful, this APT_StringMap
              object's state is replaced by the parsed string map
              description.
   throws     APT_ParseError: trouble parsing the supplied input as a
                string map declaration.  If the err argument is 0, then
                the error is thrown; if err is non-null, then the
                error object is copied into the object pointed to by
                err.
   requires   string arg non-null
*/

void setDefaultValue(APT_Int16 val);
APT_Int16 defaultValue() const;
/*
   effect     Accesses the numeric value to be used for
              unrecognized strings.
              Initially, the defaultValue() is zero.
   note       For mapString() conversion.
*/

void setDefaultString(const char*);
APT_String defaultString() const;
/*
   effect     Accesses the string to be used for unrecognized
              numeric values.
              Initially, the defaultString() is the empty string.
   note       For mapValue() conversion.
   requires   Setter arg non-null.
*/

void addAssociation(APT_Int16 val, const char* string);
/*
   effect     Adds an association between the indicated numeric
              value and the indicated string value.
              No effect if val and string are already associated.
   note       Any given numeric value may have multiple associated
              strings; any of these strings will be mapped to the
              given numeric value when using mapString().
```

```
              Similarly, any given string value can have multiple
              associated numeric values; any of these numbers will
              be mapped to the given string when using mapValue().
              When using mapString(), if the string has multiple
              associated numeric values, the first numeric value
              associated with the string is used.
              Similarly, when using mapValue(), if the numeric
              value has multiple associated strings, the first
              string associated with the numeric value is used.
   requires   string non-null.
*/

APT_Int16 mapString(const char*) const;
APT_Int16 mapString(const char*, APT_UInt32 len) const;
/*
   effect     Returns the numeric value corresponding to the given
              string.
   requires   arg non-null.
              In the first overload form, the string must be null-
              terminated.
*/

APT_String mapValue(APT_Int16 val) const;
/*
   effect     Returns the string value corresponding to the given
              numeric value.
*/

void parse(const char*, APT_ParseError* err=0);
void parse(const APT_String&, APT_ParseError* err=0);
/*
   effect     Parses the input, expecting a string map description
              (*not* starting with the token 'stringmap').
              If the parse is successful, this APT_StringMap
              object's state is replaced by the parsed string map
              description.
   throws     APT_ParseError: trouble parsing the supplied input as a
                string map declaration.  If the err argument is 0, then
                the error is thrown; if err is non-null, then the
                error object is copied into the object pointed to by
                err.
              If an error occurs, this APT_StringMap object's
              state is unchanged.
   requires   string arg non-null
*/

void unparse(ostream&) const;
APT_String unparse() const;
/*
   effect     Prints or returns a parse()able representation of
```

```
            this APT_StringMap.
*/


// state observability for testing

CaseSensitivity caseSensitivity() const;

int numValues() const;
APT_Int16 value(int index) const;
/*
   effect    Provides access to the various numeric values in the
             APT_StringMap.  The ordering of the numeric values is
             the order in which they were first added.
   requires  0 <= index < numValues()
*/

int numStrings() const;
APT_String string(int index) const;
/*
   effect    Provides access to the various string values in the
             APT_StringMap.  The ordering of the string values is
             the order in which they were first added.
   requires  0 <= index < numStrings()
*/

int numValuesForString(const char* str) const;
const APT_Int16* valuesForString(const char* str) const;
/*
   effect    Provides access to the values corresponding to the
             given string value.  The returned array is ordered
             according to order in which numeric values were
             assigned to the given string value.
             If numValues(str) is zero, returns null.
   note      The returned pointer may be subscripted from 0 to
             numValues(val)-1.
             The caller must not delete the returned pointer.
   requires  arg non-null.
*/

int numStringsForValue(APT_Int16 val) const;
const APT_String* stringsForValue(APT_Int16 val) const;
/*
   effect    Provides access to the strings corresponding to the
             given numeric value.  The returned array is ordered
             according to order in which strings were assigned to
             the given numeric value.
             If numStrings(val) is zero, returns null.
   note      The returned pointer may be subscripted from 0 to
             numStrings(val)-1.
```

```
              The caller must not delete the returned pointer.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_StringMap&);

private:
  friend class APT_Conversion;
  void parse_(APT_Lexer&, APT_ParseError* err);

  CaseSensitivity case_;
  APT_Int16 defValue_;
  APT_String defString_;

  /* Implementation assumes that the number of entries is smallish;
     if this turns out to be untrue then we'll want to go to a
     faster (hash-based?) lookup scheme. */

public:
  struct Value
  {
    APT_Int16 value_;

    APT_Int32 numStrings_;
    APT_String* strings_;
    APT_String* stringsCompare_;

    void add(const char*, CaseSensitivity); // no effect if already present

    Value();
    Value(const Value&);
    Value& operator= (const Value&);
    ~Value();

    friend APT_Archive& operator|| (APT_Archive&, Value&);
  };
private:
  APT_Int32 numValues_;
  Value* values_;

public:
  struct String
  {
    APT_String string_;
    APT_String stringCompare_;

    APT_Int32 numValues_;
    APT_Int16* values_;

    void add(APT_Int16);              // no effect if already present
```

```
    String();
    String(const String&);
    String& operator= (const String&);
    ~String();

    friend APT_Archive& operator|| (APT_Archive&, String&);
  };
private:
  APT_Int32 numStrings_;
  String* strings_;
};  // APT_StringMap
APT_DIRECTIONAL_SERIALIZATION(APT_StringMap);

class APT_Conversion
// this entire class is DEPRECATED
{
public:
  APT_Conversion();
  /*
     effect    Constructs to kind()==eDefault.
               One of the various setXXX() functions can be called to
               request a particular kind of conversion.
     DEPRECATED
  */

  APT_Conversion(const char*, APT_ParseError* err=0);
  APT_Conversion(const APT_String&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a conversion description.
               If the parse is successful, this APT_Conversion
               object's state is replaced by the parsed conversion
               description.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 conversion declaration.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
     requires  string arg non-null
     DEPRECATED
  */

  // default copy, assign, dtor OK

  void setDefault();
  /*
     effect    Converts numbers from one type to another, and between
               strings and numbers.  If the conversion is potentially
               unsafe (can lose precision or range), a warning is
               issued at step-check time.
     DEPRECATED
```

```
 */

 void setNoWarn();
 /*
    effect    Like setDefault(), except the warning (if any) about
              the conversion being potentially unsafe is supressed.
    DEPRECATED
 */

 void setNumericString(const char* format);
 /* conversion of strings and built-in numeric types (integer,
    floating point); supplied format string passed to sprintf/sscanf
    which perform the conversion.
    requires: format non-null
              format must be a valid sprintf/sscanf format string
    TBD: take an arg specifying value to be used if sscanf fails?
    DEPRECATED
 */

 void setStringMap(const APT_StringMap&);
 /* conversion of strings and numeric values via lookup table. */
 //   DEPRECATED

 void setSubString(APT_UInt32 startPos, APT_UInt32 length);
 /* conversion of longer string to shorter string via substring
    extraction */
 //   DEPRECATED

 // TBD: function callback conversion?

 enum Kind
 {
   eDefault,                  /* numeric<->numeric and string<->numeric
                                 conversion with step-check warning if
                                 potentially unsafe. */
   eNoWarn,                   /* numeric<->numeric and string<->numeric
                                 conversion with no warning. */
   eNumericString,
   eStringMap,
   eSubString
 };
 Kind kind() const;
 /*
    effect    Tells what kind of conversion this is.
    DEPRECATED
 */

 // state observers; kind() must be appropriate.

 APT_String numericStringFormat() const;
```

```
  APT_StringMap stringMap() const;
  // DEPRECATED

  void subString(APT_UInt32* startPos, APT_UInt32* length) const;
  // modifies args if non-null
  // DEPRECATED

  void parse(const char*, APT_ParseError* err=0);
  void parse(const APT_String&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a conversion description.
               If the parse is successful, this APT_Conversion
               object's state is replaced by the parsed conversion
               description.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 conversion declaration.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
               If an error occurs, this APT_Conversion object's
               state is unchanged.
     requires  string arg non-null
     DEPRECATED
  */

  void unparse(ostream&) const;
  APT_String unparse() const;
  /*
     effect    Prints or returns a parse()able representation of
               this APT_Conversion.
     DEPRECATED
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_Conversion&);
  // DEPRECATED

private:
  friend class APT_ViewAdapter;
  friend class APT_ModifyAdapter;
  void parse_(APT_Lexer&, APT_ParseError* err);

  Kind kind_;
  APT_String format_;
  APT_StringMap stringMap_;
  APT_Int32 startPos_;
  APT_Int32 length_;
};  // APT_Conversion
APT_DIRECTIONAL_SERIALIZATION(APT_Conversion);

class APT_AdapterBase
```

```
/* common base class for APT_ViewAdapter and APT_ModifyAdapter, which are
   now very nearly identical in interface. */
{
protected:
  APT_AdapterBase();

public:
  APT_AdapterBase(const APT_AdapterBase&);
  APT_AdapterBase& operator= (const APT_AdapterBase&);
  ~APT_AdapterBase();

  void setQuiet(bool q);
  bool isQuiet() const;
  /*
     effect    Manipulates a flag telling whether the framework should
               issue check-time and run-time warnings relating to
               conversions on this adapter's interface.
               Defautl is false (warn).
  */

  enum NullHandling { eHandle, eMake, eNone };
  enum Nullable { eNullable, eNotNullable, eDefault };

  void addBinding(const APT_FieldSelector& source,
                  const APT_FieldSelector& dest,
                  const APT_FieldConversion* cvt=0,
                  const APT_FieldTypeDescriptor* type=0,
                  Nullable fn=eDefault,
                  NullHandling nh=eNone,
                  const char* val=0);
  /*
     effect    Establishes a binding between the indicated source
               field and destination field.
               By default, such fields are matched by name within
               scope.
               One or more addBinding()s can be specified in an
               adapter to establish bindings with non-matching names,
               and/or to specify other aspects of the binding.
     options   cvt: specifies a conversion to be used.
                 The client retains ownership of the original cvt
                 object.
               type: specifies the type of the destination field.  For
                 a view adapter, a type specified here overrides an
                 input interface "generic" type; if the input
                 interface already has a type, the adapter-specified
                 type is ignored.  For a modify adapter, a type can be
                 specified only if the output data set does not yet
                 have a schema.
                 The client retains ownership of the original type
                 object.
```

```
            fn: specifies the nullability of the destination
              field.  For a view adapter, this can be specified
              only if the interface field's nullability matches or
              has been declared as nullable; for a modify adapter,
              this can be specified only if the output data set
              does not yet have a schema.
            nh,val: specifies whether null quelling/flagging should
               be performed for this binding.  If so, val is treated
              as a literal of the destination type, or can be
              omitted to use the default in-band null value for the
              destination type.
   note     If this binding is for a subrec or tagged, then the
            optional parameters should not be supplied.
   requires The source and dest args must not refer to a schema
            variable.
            The source and dest args must not be subscripted.
            The source field must not have been dropped via
            drop().
            The source and dest fields must either both be
            top-level fields, or must appear within scopes that are
            associated (by name or explicit binding).
            The destination field must not already have a binding.
            The source and destination fields must have equal
            vector status (both scalar, both var-length vectors, or
            both fixed-length vectors of equal length).
            If type is 0, then fn must be eDefault.
            If nh is eNone, then val must be 0.
            If non-null, cvt->isDefault() must be false.
*/

bool lookupBinding(APT_FieldSelector* source,
                   APT_FieldSelector* dest,
                   APT_FieldConversion* *cvt,
                   APT_FieldTypeDescriptor* *type,
                   Nullable* fn,
                   NullHandling* nh,
                   APT_String* val,
                   int which=0,
                   bool* hasVal=0) const;
/*
   effect   Retrieves the binding associated with the given
            source or destination field.
            If source is non-empty, the lookup is by source field
            yielding destination field.  If there are multiple
            destinations bound to a given source, the first binding
            is returned unless a non-zero "which" is specified, in
            which case the indicated binding is returned.
            If dest is non-empty, the lookup is by dest field
            yielding source field.
            If there is a binding, modifies the empty field
```

```
            selector argument and the other args (if non-null), and
            returns true.
            If there is no binding, does not modify the arguments
            and returns false.
   note     If cvt is non-null, the pointer to which it points will
            be set to point to a field conversion object (or null).
            The client is responsible for disOwn()ing the field
            conversion object.
            Similarly for the type return parameter.
            If addBinding() was called with a val==0 argument, then
            lookupBinding()'s val return parameter will be set to the
            empty string.  To disambiguate this from the case where
            addBinding() was called with val=="", the optional hasVal
            return parameter can be supplied.
   requires source, dest non-null.
            One of source and dest must be the empty field
            selector.
            which may be non-zero only if source is non-empty.
*/

void drop(const APT_FieldSelector& source);
/*
   effect   Specifies that the indicated item should be dropped.
   requires source must not refer to a schema variable.
            source must not be subscripted.
            source must not already have been dropped.
            source must not have been bound via addBinding()
            source must not be a case of a tagged aggregate.
            For a view adapter, a dropped field must not appear
            in the interface.
            For a modify adapter, the output data set must either
            not yet have a schema, or the schema must not contain
            the dropped field.
            drop() and keep() are mutually exclusive.
*/
bool isDropped(const APT_FieldSelector& source) const;
/*
   effect   Tells if drop() has been called for source.
            Alternately, if any keep()s have been performed, and
            source is not among the keep()s or bindings of this
            adapter, then the field is also considered to have been
            dropped by this adapter.
*/

void keep(const APT_FieldSelector& source);
/*
   effect   Specifies that the indicated item should be kept.
   requires source must not refer to a schema variable.
            source must not be subscripted.
            source must not already have been kept.
```

```
            For a modify adapter, the output data set must either
            not yet have a schema, or the data set schema must contain
            the kept field.
            drop() and keep() are mutually exclusive.
*/
bool isKept(const APT_FieldSelector& source) const;
/*
   effect   Tells if keep() has been called for source.
*/

void parse(const char*, APT_ParseError* err=0);
void parse(const APT_String&, APT_ParseError* err=0);
void parse(istream&, APT_ParseError* err=0);
/*
   effect   Parses the input, expecting an adapter description.
            If the parse is successful, this APT_AdapterBase
            object's state is replaced by the parsed adapter
            description.
   throws   APT_ParseError: trouble parsing the supplied input as an
              adapter declaration.  If the err argument is 0, then
              the error is thrown; if err is non-null, then the
              error object is copied into the object pointed to by
              err.
            If an error occurs, this APT_AdapterBase object's
            state is unchanged.
   requires  string arg non-null
*/

void unparse(ostream&) const;
APT_String unparse() const;
/*
   effect   Prints or returns a parse()able representation of
            this APT_AdapterBase.
*/

// full state observability; mostly for testing

int numBindings() const;
void binding(int index,
            APT_FieldSelector* source,
            APT_FieldSelector* dest,
            APT_FieldConversion* *cvt,
            APT_FieldTypeDescriptor* *type,
            Nullable* fn,
            NullHandling* nh,
            APT_String* val,
            bool* hasVal=0) const;
// requires: 0 <= index < numBindings()

int numDropped() const;
```

```
  APT_FieldSelector dropped(int index) const;
  // requires: 0 <= index < numDropped()

  int numKept() const;
  APT_FieldSelector kept(int index) const;
  // requires: 0 <= index < numKept()

  friend APT_Archive& operator|| (APT_Archive&, APT_AdapterBase&);

  void parse_(APT_Lexer&, APT_ParseError* err);

private:
  APT_AdapterRep* rep_;
};
APT_DIRECTIONAL_SERIALIZATION(APT_AdapterBase);

class APT_ViewAdapter : public APT_AdapterBase
// the class formerly known as APT_InputAdapter...
{
public:
  APT_ViewAdapter();
  /*
     effect    Makes an empty view adapter.
  */

  APT_ViewAdapter(const APT_ViewAdapter&);
  APT_ViewAdapter& operator= (const APT_ViewAdapter&);
  ~APT_ViewAdapter();

  APT_ViewAdapter(const char*, APT_ParseError* err=0);
  APT_ViewAdapter(const APT_String&, APT_ParseError* err=0);
  APT_ViewAdapter(istream&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a view adapter
               description.
               If the parse is successful, this APT_ViewAdapter
               object's state is replaced by the parsed view adapter
               description.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 view adapter declaration.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
     requires  string arg non-null
  */

  void bind(const APT_FieldSelector& dsField,
            const APT_FieldSelector& interfaceField,
            const APT_FieldConversion* cvt=0);
  // DEPRECATED
```

```
/*
   effect    Identifies which dataset field will be bound to an
             interface schema input field.
             By default, such fields are matched by name within
             context, with potential matches being type-qualified.
             One or more bind()s can be specified in an
             adapter to establish bindings with non-matching names,
             and/or to specify an APT_FieldConversion.
   note      The client retains ownership of the cvt argument, if
             any.
   requires  interfaceField must not refer to a schema variable.
             interfaceField must not refer to an aggregate.
             interfaceField must not be subscripted.
             dsField may not be subscripted.
             interfaceField must not already be bound.
             If interfaceField is within an aggregate that has been
             bound to a dataset aggregate, then dsField must refer
             to a top-level component among the fields of the
             concrete aggregate to which the interfaceField's
             aggregate is bound.
             If multiple fields of a tagged interface aggregate are
             bound, they must be bound to mutually exclusive
             (members of a single tagged aggregate) dataset fields.
             If interfaceField is not a vector, then dsField must
             not be a vector.
             If interfaceField is a vector, then dsField must be a
             vector.
             If non-null, cvt->isDefault() must be false.
*/

bool lookupFieldBinding(APT_FieldSelector* intfField,
                        APT_FieldSelector* dsField,
                        APT_FieldConversion* *cvt) const;
// DEPRECATED
/*
   effect    Retrieves the binding associated with the given
             interface or dataset field.
             If intfField is non-empty, the lookup is by interface
             field yielding dataset field.
             If dsField is non-empty, the lookup is by dataset
             field yielding interface field.
             If there is a binding, modifies the empty field
             selector argument and the cvt arg, and returns true.
             If there is no binding, does not modify the arguments
             and returns false.
   note      If cvt is non-null, the pointer to which it points will
             be set to point to a field conversion object (or null).
             The client is responsible for deleting the field
             conversion object (if its canParameterize() is true).
   requires  intfField non-null.
```

```
            dsField non-null.
            One of intfField and dsField must be the empty field
            selector.
*/

void bindAggregate(const APT_FieldSelector& dsAggr,
                   const APT_FieldSelector& interfaceAggr);
// DEPRECATED
/*
   effect    Identifies which dataset aggregate will be bound to an
             interface schema aggregate.
             The following bindings are accepted:

             DS aggregate   Interface aggregate   Legal   Notes
             ------------   -------------------   -----   -----
                subrec            subrec           yes
                subrec            tagged           no

                tagged            subrec           no
                tagged            tagged           yes

   requires  interfaceAggr must not refer to an ordinary field.
             interfaceAggr must not refer to a schema variable.
             interfaceAggr must not be subscripted.
             dsAggr must not be subscripted (TBD: even not its final
             component?)
             interfaceAggr must not already be bound.
             If interfaceAggr is within an aggregate that has been
             bound to a dataset aggregate, then dsAggr must refer to
             a top-level component among the fields of the concrete
             aggregate to which the interfaceAggr's aggregate is
             bound.
             If interfaceAggr is not a vector, then dsAggr must
             not be a vector.
             If interfaceAggr is a vector, then dsAggr must be a
             vector.
*/

bool lookupAggrBinding(APT_FieldSelector* intfAggr,
                       APT_FieldSelector* dsAggr) const;
// DEPRECATED
/*
   effect    Retrieves the binding associated with the given
             interface or dataset aggregate field.
             If intfField is non-empty, the lookup is by interface
             aggregate yielding dataset aggregate.
             If dsField is non-empty, the lookup is by dataset
             aggregate yielding interface aggregate.
             If there is a binding, modifies the empty field
             selector argument, and returns true.
```

```
              If there is no binding, does not modify the arguments
              and returns false.
   requires   intfAggr non-null.
              dsAggr non-null.
              One of intfAggr and dsAggr must be the empty field
              selector.
*/

void copyBinding(const APT_ViewAdapter* source,
                 const APT_FieldSelector& sourceIntf,
                 const APT_FieldSelector& var="",
                 const APT_TransferAdapter* ta=0,
                 const APT_FieldSelector& destIntf="");
/*
   effect     Examines the given source adapter to see if it has a
              field (or aggregate) binding for the indicated
              sourceIntf component.
              If not, there is no effect.
              If the source adapter does have a binding for the
              indicated sourceIntf component, then a corresponding
              binding is added to this view adapter:
              - the source adapter's source field (and conversion, if
                any) are used.
              - if a var and transfer adapter are provided, then the
                source adapter is examined to see if a mapVar() is
                renaming the source field in a transfer; if so, this
                is taken into account: the binding added to this
                view adapter will have the renamed field as the
                field binding's source field.
              - if a destIntf is provided, then the added binding
                will use it instead of sourceIntf.
*/

// full state observability; mostly for testing

int numFieldBindings() const;  // DEPRECATED
void fieldBinding(int index, APT_FieldSelector* dsField,
                  APT_FieldSelector* intfField,
                  APT_FieldConversion* *cvt) const;  // DEPRECATED

// deprecated interface; see APT_TransferAdapter
enum VarBinding { eAll, eSpecific };
void setVarMode(const APT_FieldSelector& interfaceVar, VarBinding vb);
void addToVar(const APT_FieldSelector& interfaceVar, const char* field);
void dropFromVar(const APT_FieldSelector& interfaceVar, const char* field);
void mapVar(const APT_FieldSelector&, const char*, const char*);
APT_TransferAdapter transferAdapter() const;
void setTransferAdapter(const APT_TransferAdapter&);

APT_Schema viewAdaptedSchema(const APT_Schema& src) const;
```

```
  APT_Schema inputAdaptedSchema(const APT_Schema& src) const; // old name
  /*
     effect    This "projects" the given schema through this view
               adapter, producing a new schema where:
               - fields might have been dropped by the adapter
               - fields might have been renamed by the adapter
               - fields might have been retyped and/or converted by
                 the adapter
               - fields might have had their nullability changed via
                 explicit nullable or not_nullable (default is to
                 inherit source field's nullability).  Nullability can
                 also be changed via make_null, handle_null, null, and
                 not_null.
     note      Any intact sub-records of the src schema are first
               expanded.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_ViewAdapter&);

private:
  APT_TransferAdapter* ta_;     // to support the deprecated interface
};
APT_DIRECTIONAL_SERIALIZATION(APT_ViewAdapter);

// back-compatibility
typedef APT_ViewAdapter APT_InputAdapter;

class APT_ModifyAdapter : public APT_AdapterBase
// the class formerly known as APT_OutputAdapter...
{
public:
  APT_ModifyAdapter();
  /*
     effect    Makes an empty modify adapter.
  */

  APT_ModifyAdapter(const APT_ModifyAdapter&);
  APT_ModifyAdapter& operator= (const APT_ModifyAdapter&);
  ~APT_ModifyAdapter();

  APT_ModifyAdapter(const char*, APT_ParseError* err=0);
  APT_ModifyAdapter(const APT_String&, APT_ParseError* err=0);
  APT_ModifyAdapter(istream&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a modify adapter
               description.
               If the parse is successful, this APT_ModifyAdapter
               object's state is replaced by the parsed modify adapter
               description.
     throws    APT_ParseError: trouble parsing the supplied input as a
```

```
                modify adapter declaration.  If the err argument is 0, then
                the error is thrown; if err is non-null, then the
                error object is copied into the object pointed to by
                err.
   requires  string arg non-null
*/

void dropField(const APT_FieldSelector& interfaceFieldOrAggr);
// DEPRECATED
/*
   effect    Specifies that the indicated interface schema output
             field or aggregate should not be carried in the output
             dataset.
   requires  interfaceFieldOrAggr must not refer to a schema variable.
             interfaceFieldOrAggr must not be subscripted.
             interfaceFieldOrAggr must not have been bound via
             bind() or bindAggregate().
             This call can be used only if the dataset to which
             the binding is being made has no schema (its schema is
             being determined by the operator output).
*/

bool isFieldDropped(const APT_FieldSelector& interfaceFieldOrAggr) const;
// DEPRECATED
/*
   effect    Tells if dropField() has been called for
             interfaceFieldOrAggr.
*/

void bind(const APT_FieldSelector& interfaceField,
          const APT_FieldSelector& dsField,
          const char* type=0,
          const APT_FieldConversion* cvt=0);
// DEPRECATED
/*
   effect    Identifies which dataset field will be bound to an
             interface schema output field.
             By default, such fields are matched by name within
             context, with potential matches being type-qualified.
             One or more bind()s can be specified in an
             adapter to establish bindings with non-matching names,
             to specify destination type, and/or to specify an
             APT_FieldConversion.
   note      The client retains ownership of the cvt argument, if
             any.
   requires  interfaceField must not refer to a schema variable.
             interfaceField must not refer to an aggregate.
             interfaceField must not be subscripted.
             dsField may not be subscripted.
             dsField must not already be bound.
```

```
            interfaceField must not already be bound.
            interfaceField (or any of its parent scopes) must not
            have been dropped via dropField().
            If interfaceField is within an aggregate that has been
            bound to a dataset aggregate, then dsField must refer
            to a top-level component among the fields of the
            concrete aggregate to which the interfaceField's
            aggregate is bound.
            If multiple fields of a tagged interface aggregate are
            bound, they must be bound to mutually exclusive
            (members of a single tagged aggregate) dataset fields.
            If interfaceField is not a vector, then dsField must
            not be a vector.
            If interfaceField is a vector, then dsField must be a
            vector.
            The type arg may be non-null only if the dataset to which
            the binding is being made has no schema (its schema is
            being determined by the operator output).  type may not
            be an aggregate type.
            If provided, type must lookup and parse to a valid
            non-aggr, non-var APT_FieldTypeDescriptor.
            If non-null, cvt->isDefault() must be false.
*/

bool lookupFieldBinding(APT_FieldSelector* intfField,
                        APT_FieldSelector* dsField,
                        APT_String* type,
                        APT_FieldConversion* *cvt) const;
// DEPRECATED
/*
   effect    Retrieves the binding associated with the given
             interface or dataset field.
             If intfField is non-empty, the lookup is by interface
             field yielding dataset field.
             If dsField is non-empty, the lookup is by dataset
             field yielding interface field.
             If there is a binding, modifies the empty field
             selector argument and the type and cvt args, and returns
             true.
             If there is no binding, does not modify the arguments
             and returns false.
   note      If cvt is non-null, the pointer to which it points will
             be set to point to a field conversion object (or null).
             The client is responsible for deleting the field
             conversion object (if its canParameterize() is true).
   requires  intfField non-null.
             dsField non-null.
             One of intfField and dsField must be the empty field
             selector.
*/
```

```
void bindAggregate(const APT_FieldSelector& interfaceAggr,
                   const APT_FieldSelector& dsAggr);
// DEPRECATED
/*
   effect    Identifies which dataset aggregate field will be bound
             to an interface schema aggregate output field.
             The following bindings are accepted:

             Interface aggregate   DS aggregate   Legal   Notes
             -------------------   ------------   -----   -----
                   subrec             subrec       yes
                   subrec             tagged       no

                   tagged             subrec       no
                   tagged             tagged       yes

   requires  interfaceAggr must not refer to a schema variable.
             interfaceAggr must not refer to an ordinary field.
             interfaceAggr must not be subscripted.
             dsAggr must not be subscripted (TBD: even not its final
             component?)
             dsAggr must not already be bound.
             interfaceAggr (or any of its parent scopes) must not
             have been dropped via dropField().
             If interfaceAggr is within an aggregate that has been
             bound to a dataset aggregate, then dsAggr must refer to
             a top-level component among the fields of the concrete
             aggregate to which the interfaceAggr's aggregate is
             bound.
             If interfaceAggr is not a vector, then dsAggr must
             not be a vector.
             If interfaceAggr is a vector, then dsAggr must be a
             vector.
*/

bool lookupAggrBinding(APT_FieldSelector* intfAggr,
                       APT_FieldSelector* dsAggr) const;
// DEPRECATED
/*
   effect    Retrieves the binding associated with the given
             interface or dataset aggregate field.
             If intfField is non-empty, the lookup is by interface
             aggregate yielding dataset aggregate.
             If dsField is non-empty, the lookup is by dataset
             aggregate yielding interface aggregate.
             If there is a binding, modifies the empty field
             selector argument, and returns true.
             If there is no binding, does not modify the arguments
             and returns false.
```

```
     requires  intfAggr non-null.
               dsAggr non-null.
               One of intfAggr and dsAggr must be the empty field
               selector.
  */

  void copyBinding(const APT_ModifyAdapter* source,
                   const APT_FieldSelector& sourceIntf,
                   const APT_FieldSelector& destIntf="");
  /*
     effect    Examines the given source adapter to see if it has a
               field (or aggregate) binding for the indicated
               sourceIntf component, or if it has a drop.
               If not, there is no effect.
               If the source adapter drops the sourceIntf field or
               aggregate, then a corresponding drop is added to this
               modify adapter (using destIntf, if supplied).
               If the source adapter has a binding for the indicated
               sourceIntf component, then a corresponding binding is
               added to this modify adapter:
               - the source adapter's source field (and
                 typespec/conversion, if any) are used.
               - if a destIntf is provided, then the added binding
                 will use it instead of sourceIntf.
  */

  // full state observability, mostly for testing

  int numDroppedFields() const;  // DEPRECATED
  APT_FieldSelector droppedField(int index) const;  // DEPRECATED

  int numFieldBindings() const;  // DEPRECATED
  void fieldBinding(int index, APT_FieldSelector* interfaceField,
                    APT_FieldSelector* dsField,
                    APT_String* type,
                    APT_FieldConversion* *cvt) const;  // DEPRECATED

  friend APT_Archive& operator|| (APT_Archive&, APT_ModifyAdapter&);

//  void parse_(APT_Lexer&, APT_ParseError* err);
};
APT_DIRECTIONAL_SERIALIZATION(APT_ModifyAdapter);

// back-compatibility
typedef APT_ModifyAdapter APT_OutputAdapter;


class APT_TransferAdapter
{
public:
```

```
APT_TransferAdapter();
/*
   effect    Makes an empty transfer adapter.
*/

APT_TransferAdapter(const APT_TransferAdapter&);
APT_TransferAdapter& operator= (const APT_TransferAdapter&);
~APT_TransferAdapter();

APT_TransferAdapter(const char*, APT_ParseError* err=0);
APT_TransferAdapter(const APT_String&, APT_ParseError* err=0);
APT_TransferAdapter(istream&, APT_ParseError* err=0);
/*
   effect    Parses the input, expecting a transfer adapter
             description.
             If the parse is successful, this APT_TransferAdapter
             object's state is replaced by the parsed transfer adapter
             description.
   throws    APT_ParseError: trouble parsing the supplied input as a
               transfer adapter declaration.  If the err argument is 0, then
               the error is thrown; if err is non-null, then the
               error object is copied into the object pointed to by
               err.
   requires  string arg non-null
*/

enum VarBinding
{
  eAll,
  eSpecific
};

void setVarMode(const APT_FieldSelector& interfaceVar, VarBinding vb);
/*
   effect    Causes the indicated schema variable to be bound to:
               eAll: all fields in the schema variable's scope.  See
               dropFromVar().
               eSpecific: no fields.  See addToVar().
   note      By default, schema variables bind to eAll.
   requires  interfaceVar must refer to a schema variable.
             interfaceVar must not be subscripted.
             Once addToVar() or dropFromVar() is called on a
             particular interfaceVar, setVarMode() may not change
             the mode of the interfaceVar from that established by
             the addToVar() or dropFromVar() calls.
*/

void addToVar(const APT_FieldSelector& interfaceVar,
              const char* field);
/*
```

```
   effect    Adds the indicated field to the indicated schema
             variable's binding.
             No effect if field is already bound to the schema
             variable.
   note      The varMode() for interfaceVar is set to eSpecific.
   requires  interfaceVar must refer to a schema variable.
             If setVarMode() was called for interfaceVar, its mode
             must have been set to eSpecific.
             No dropFromVar() calls may have been made on
             interfaceVar.
             interfaceVar must not be subscripted.
             field non-null.
             field must be a top-level component in var's context
*/

void dropFromVar(const APT_FieldSelector& interfaceVar,
                 const char* field);
/*
   effect    Removes the indicated field from the indicated schema
             variable's binding.
   note      The varMode() for interfaceVar is set to eAll.
   requires  interfaceVar must refer to a schema variable.
             If setVarMode() was called for interfaceVar, its mode
             must have been set to eAll.
             No addToVar() calls may have been made on interfaceVar.
             interfaceVar must not be subscripted.
             field non-null.
             field must be a top-level component in var's context
*/

void renameFieldInVar(const APT_FieldSelector& interfaceVar,
                      const char* field, const char* newName);
/*
   effect    Specifies a renaming to be applied to a field bound to
             the indicated schema variable.  The renaming is
             performed whenever a APT_Operator::transfer() is
             performed with interfaceVar as a participant.
   requires  interfaceVar must refer to a schema variable.
             interfaceVar must not be subscripted.
             field non-null.
             field must not have been dropFromVar()ed.
             field must be a part of interfaceVar's binding (after any
             adds, but before any renameFieldInVar()s).
             field must not already have been renameFieldInVar()ed.
             field must not be subscripted.
             newName non-null.
             newName unique (after all renames are applied).
             newName must be a valid field identifier.
             newName must not be subscripted.
*/
```

```
void parse(const char*, APT_ParseError* err=0);
void parse(const APT_String&, APT_ParseError* err=0);
void parse(istream&, APT_ParseError* err=0);
/*
   effect    Parses the input, expecting a transfer adapter
             description.
             If the parse is successful, this APT_TransferAdapter
             object's state is replaced by the parsed transfer adapter
             description.
   throws    APT_ParseError: trouble parsing the supplied input as a
               transfer adapter declaration.  If the err argument is 0, then
               the error is thrown; if err is non-null, then the
               error object is copied into the object pointed to by
               err.
             If an error occurs, this APT_TransferAdapter object's
             state is unchanged.
   requires  string arg non-null
*/

void unparse(ostream&) const;
APT_String unparse() const;
/*
   effect    Prints or returns a parse()able representation of
             this APT_TransferAdapter.
*/


// full state observability; mostly for testing

int numVars() const;
APT_FieldSelector getVar(int) const;
bool hasVar(const APT_FieldSelector& interfaceVar) const;

VarBinding varMode(const APT_FieldSelector& interfaceVar) const;
// requires: hasVar(interfaceVar) is true

bool addedToVar(const APT_FieldSelector& intfVar, const char* field) const;
bool droppedFromVar(const APT_FieldSelector& intfVar,
                    const char* field) const;
APT_String varFieldRenamedTo(const APT_FieldSelector&,
                                  const char* field) const;
// requires: hasVar(interfaceVar) is true

int numAddedToVar(const APT_FieldSelector& interfaceVar) const;
APT_String addedVarField(const APT_FieldSelector& interfaceVar,
                                 int index) const;
// requires: hasVar(interfaceVar) is true

int numDroppedFromVar(const APT_FieldSelector& interfaceVar) const;
```

```
  APT_String droppedVarField(const APT_FieldSelector& interfaceVar,
                             int index) const;
  // requires: hasVar(interfaceVar) is true

  int numVarRenames(const APT_FieldSelector& interfaceVar) const;
  void varRename(const APT_FieldSelector& interfaceVar, int index,
                 APT_String* field,
                 APT_String* newName) const;
  // requires: hasVar(interfaceVar) is true

  friend APT_Archive& operator|| (APT_Archive&, APT_TransferAdapter&);

  void parse_(APT_Lexer&, APT_ParseError* err);

  struct IntfVariableDesc;      // forward declare

private:
  const IntfVariableDesc* validateVar(const APT_FieldSelector& intfVar) const;
  IntfVariableDesc* validateVar(const APT_FieldSelector& intfVar);
  /*
     effect    Returns a pointer to the entry corresponding to
               intfVar.  An entry is created if necessary.
     requires  interfaceVar must not refer to an ordinary field.
               interfaceVar must not refer to an aggregate.
               interfaceVar must not be subscripted.
  */

public:
  struct IntfVariableDesc
  {
    APT_FieldSelector intfVar_;
    VarBinding mode_;
    bool modeSet_;

    APT_Int32 numAddedFields_;
    APT_FieldSelector* addedFields_;

    APT_Int32 numDroppedFields_;
    APT_FieldSelector* droppedFields_;

    struct Rename
    {
      APT_FieldSelector field_;
      APT_FieldSelector newName_; // same as field_ if not renaming

      // default ctor, copy, assign, dtor OK
      friend APT_Archive& operator|| (APT_Archive&, Rename&);
    };
    APT_Int32 numRenames_;
    Rename* renames_;
```

```
    IntfVariableDesc();
    IntfVariableDesc(const IntfVariableDesc&);
    IntfVariableDesc& operator= (const IntfVariableDesc&);
    ~IntfVariableDesc();

    friend APT_Archive& operator|| (APT_Archive&, IntfVariableDesc&);
  };
private:
  APT_Int32 numIntfVariables_;
  IntfVariableDesc* intfVariables_;
};
APT_DIRECTIONAL_SERIALIZATION(APT_TransferAdapter);


#endif // APT_ADAPTER_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_COLLECTOR_H
#define APT_COLLECTOR_H

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_ISDYNAMIC_H
#include <apt_util/isdynamic.h>
#endif

class APT_FieldSelector;
class APT_InputAccessorBase;
class APT_InputInterface;
class APT_Schema;
class APT_CollectorRep;
class APT_OperatorRep;
class APT_CollectorSC;
class APT_PropertyList;
class APT_ErrorLog;
class APT_DataSetRep;

class APT_Collector : public APT_Persistent, public APT_IsDynamic
{
  APT_DECLARE_RTTI(APT_Collector);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_Collector);

protected:
  APT_Collector();
  /*
     effect    Initializes this collector to the "undescribed" state.
               The describeCollector() function will be called by the
               framework when first needed.
  */

public:
```

```
virtual ~APT_Collector();

APT_Status initializeFromArgs(const APT_PropertyList& args);
/*
   effect    Initializes this collector's state from the indicated
             property list.
             Wraps the initializeFromArgs_() virtual (calling it in
             eInitial mode), and stores the argument property list
             in the rep (where it is serialized by the framework).
             See initializeFromArgs_() for more details.
   note      Although this is designed to be an OSL entry point,
             it can also be used from C++ ADI programs to initialize
             OSL-aware collectors via their command-line argument
             interface.  The args property list should be of the
             form produced by this collector's OSL wrapper.
*/

APT_PropertyList initializationArgs() const;
/*
   effect    Returns the property list that was passed to
             initializeFromArgs() (or setRuntimeArgs(), if that was
             called).
             Returns the empty list if neither of these functions
             was called.
*/


APT_String errorInformation() const;
/*
   effect    If reportError() has been called (or the errorLog()
             used), returns the error text.  Otherwise returns the
             empty string.
   DEPRECATED
*/

APT_String ident() const;
/*
   effect    Returns a string by which this collector may be
             identified.
             If setIdent() has not been called (or has been called
             with an empty string arg), returns the class name of
             this collector.
             ident() is used to distinguish between multiple instances
             of the same collector in a step.
*/
void setIdent(const char*);
/*
   effect    Determines what ident() returns.
```

```
    requires  Arg non null.
  */

protected:
  enum InitializeContext
  {
    eInitial,           /* on conductor, prior to describeCollector()
                           and storing serialization */
    eRun                /* on player(s), after loading serialization
                           but prior to setupInputs() */
  };
  virtual APT_Status initializeFromArgs_(const APT_PropertyList& args,
                                         InitializeContext context);
  /*
     effect    If an collector supports OSL, it should override this
               function to interpret the supplied arguments.
               Any errors should be reported via reportError() and a
               APT_StatusFailed return value.
               The default implementation returns APT_StatusFailed and
               writes "Not an OSH-enabled collector" using
               reportError().
     note      This function is called twice: once on the conductor
               before describeCollector(), and then in the player(s)
               before runLocally().  The context argument can be used
               to distinguish between the two calls.
               Both calls receive the same property list (unless
               setRuntimeArgs() is called by this function or
               describeCollector()).
     returns   APT_StatusOk: this collector is happy with its arguments
               APT_StatusFailed: this collector found a problem with
                 its arguments; more information may be supplied via
                 reportError().
  */

  virtual APT_Status describeCollector()=0;
  /*
     effect    Must be overridden to describe this collector's
               interface, by calling setInputInterfaceSchema().
               Note that a collector can have no interface schema,
               in which case describeCollector() can be overridden
               to do nothing.
     note      This function is called once by the framework at
               step-check time.
               The this collector's operator's describeOperator()
               function will have been called already.
     return    APT_StatusOk: All seems to be in order.
               APT_StatusFailed: Something is amiss.  See reportError().
  */
```

```
virtual APT_Status setupInputs(int numPartitions)=0;
/*
   effect    Must be overridden to setup this collector's
             accessor(s), by calling setupInputAccessor().
             The numPartitions argument is guaranteed to be
             positive.  It is the same as what will be passed to
             selectInput().
   note      This function will be called by the framework prior to
             the first call to selectInput().
             Unlike operators, collectors typically contain their
             accessor objects as private data members, initialized
             in setupInputs() and used in selectInput().
             If this is a keyless collector, this function should
             be overridden to do nothing.
   return    APT_StatusOk: All seems to be in order.
             APT_StatusFailed: Something is amiss.  See reportError().
*/

void discardInput();
/*
   effect    If called by selectInput(), causes the record selected
             by selectInput() (indicated by the value it is about to
             return) to be discarded.
   note      This applies only to the current selectInput() call.
   requires  This routine may only be called during a selectInput()
             call.
*/

virtual int selectInput(int numPartitions)=0;
/*
   effect    Called by the framework to choose which input partition
             will supply the next record input to a sequential
             operator's input.
             This function must be overridden to examine the current
             input record of each partition (via the accessor
             objects initialized in setupInputs()) and return a
             partition number identifying .
             The numPartitions argument is guaranteed to be
             positive.  It specifies the number of partitions
             being fed into the sequential operator's input.
   note      Some of the input partitions may have reached EOF;
             numPartitions is always the total number of input
             partitions regardless of how many have reached EOF.
             Overrides can determine whether a given partition has
             reached EOF via atEOF().
             At least one partition will not be at EOF.
   requires  The returned value must satisfy:
```

```
              0 <= retval < numPartitions
            Alternately, if retval==-1, a fatal error is signalled
            by the framework (in which case it is advised that the
            collector will have placed relevant information into
            its errorLog()).
            atEOF(retval) must be false
*/


/**** functions called from describeCollector() overrides ****/

/* These functions should only be called from within
   describeCollector().  It should never be called from within
   setupInputs() or selectInput(). */

APT_Schema inputInterfaceSchema() const;
void setInputInterfaceSchema(const APT_Schema& schema);
/*
   effect    Accesses the collector-centric schema to be used for
             this collector's input records.
             The schema object is copied.
             This is optional: it is legal for a collector to have
             no interface schema.  Such collectors are called
             "keyless" collectors.
   requires  The schema must consist of zero or more non-aggregate,
             non-vector fields.  No schema variables.
             This function may only be called from within
             describeCollector(), and may only be called once.
*/

APT_Schema inputConcreteSchema() const;
/*
   effect    Returns the concrete schema associated with the
             input dataset being merged by this collector.
   note      If this collector is in an operator, this function
             returns the input interface schema of the operator.
             If this collector is attached directly to a data set,
             this function returns the concrete schema of the data
             set.
*/

APT_Schema viewAdaptedSchema() const;
/*
   effect    Returns the schema computed by taking the concrete
             schema and "projecting" it through the view adapter
             associated with this collector, if any.
             See APT_ViewAdapter::viewAdaptedSchema() for
             details.
```

```
  */

  void setRuntimeArgs(const APT_PropertyList& args);
  /*
     effect    Can be called from describeCollector() to specify an
               initialization property list to be passed to
               initializeFromArgs_() at eRun time.
     note      This is intended to be useful for the case of an
               OSL-enabled collector used via its C++ interface, where
               the collector's implementation does not use persistence
               but rather uses initializeFromArgs_() at runtime.
  */

  void reportError(const char*);
  /*
     effect    This function can optionally be called by
               describeCollector() or setupInputs() before returning
               APT_StatusFailed.
               The string argument can contain any information that
               might be useful in describing why describeCollector()
               or setupInputs() is failing.
               This function may be called multiple times; the error
               strings are appended.
     requires  arg non-null.
  */

public:
  APT_ErrorLog& errorLog();
  /*
     effect    Returns access to this collector's internal error log
               object.
     note      Any traffic generated on this log will be identified as
               originating from this collector.
  */


  /**** functions called from setupInputs() overrides ****/

  /* This function should only be called from within
     setupInputs().  It should never be called from within
     describeCollector() or selectInput(). */

protected:
  void setupInputAccessor(const APT_FieldSelector& component,
                          APT_InputAccessorBase* acc,
                          int partNum);
  /*
     effect    Sets up the indicated accessor object to access the
```

```
                indicated component of the indicated partition of the
                input.
                The component argument is evaluated relative to the
                schema that was specified in the call to
                setInputInterfaceSchema().
     requires   acc non-null
                0 <= partNum < numPartitions
                The component must not already have an accessor.
                If acc->hasType() is true, then component must identify
                a field component of the indicated dataset's interface
                schema whose type exactly matches the accessor's
                datatype.
                If acc->hasType() is false, then the accessor's type
                will be determined by the field component.
  */

  APT_InputInterface* inputInterface(int partNum);
  /*
     effect     Returns this collector's input interface object for
                the indicated partition of the input.
     requires   0 <= partNum < numPartitions
  */


  /**** functions called from selectInput() overrides ****/

  bool atEOF(int partNum) const { return eofFlags_ && eofFlags_[partNum]; }
  /*
     effect     Tells if the indicated input partition has reached
                EOF.
     requires   0 <= partNum < numPartitions
  */

private:
  friend class APT_CollectorRep;
  friend class APT_OperatorRep;
  friend class APT_DataSet;
  friend class APT_DataSetRep;
  friend APT_Archive& operator|| (APT_Archive&, APT_DataSetRep&);

  // prohibit copying
  APT_Collector(const APT_Collector&);
  APT_Collector& operator= (const APT_Collector&);

  APT_CollectorRep* rep_;
  bool* eofFlags_;
};
```

```
#endif // APT_COLLECTOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_COMPOSITE_H
#define APT_COMPOSITE_H

#ifndef APT_OPERATOR_H
#include <apt_framework/operator.h>
#endif

class APT_CompositeOperator : public APT_Operator
{
  APT_DECLARE_RTTI(APT_CompositeOperator);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_CompositeOperator);

protected:
  APT_CompositeOperator();
  // copy/assign prohibited in base class

//  virtual APT_Status describeOperator()=0;
  /* the rules for composite operators' describeOperator() override
     are different, and are summarized here */
  /*
     effect    Must be overridden to do the following:
               - setInputDataSets(), setOutputDataSets()
               - markSubOperator() for all sub-operators
               - call redirectInput() and redirectOutput() for all
                  input and output datasets.
     requires  setKind() should not be called.
               setInputInterfaceSchema(), setOutputInterfaceSchema()
               should not be called.
               declareTransfer() should not be called
               setPartitionMethod() should not be called
               setCollectionMethod() should not be called
               setCheckpointStateHandling() should not be called
  */

  virtual APT_Status runLocally(); // override
  /*
     effect    Because this is a composite operator, this function
               will not be called when the step is run.  Instead, the
               sub-operators of this composite operator will be run by
               the step.
     requires  This should not be overridden in derivations.
  */
```

```
void markSubOperator(APT_Operator* sub);
/*
   effect    Should be called by derivations' describeOperator()
             override for each of the sub-operators contained within
             the composite operator.
   note      Sub-operators are typically allocated within the
             composite operator's describeOperator() method.
             This routine attaches the sub-operator to this
             composite operator's step; the client should *not* call
             step.attachOperator() for the sub-operator.
             If setRequestedKind() has been called on this composite
             operator, the requested kind is conveyed to each
             sub-operator by markSubOperator().  The composite's
             describeOperator() should explicitly call
             setRequestedKind() for a sub-operator (before or after
             markSubOperator()) to re-assert the correct setting if
             the composite's requested kind is not appropriate for
             the sub-operator.
   requires  sub non-null
             sub must not be a sub-operator of any other composite
             operator.
             sub should be dynamically allocated.  The framework
             will delete the sub operator; the client should not
             do so.
*/

void redirectInput(int inputDS, APT_Operator* sub, int subInputDS);
/*
   effect    Specifies that the indicated input of this composite
             operator is to be tied to the indicated sub-operator's
             indicated input.
   note      The redirection is performed immediately.
             The composite operator's input adapter is moved onto
             the indicated sub-operator.
   requires  0 <= inputDS < inputDataSets()
             0 <= subInputDS < sub->inputDataSets()
             setInputDataSets() must have been called.
             sub non-null
             markSubOperator(sub) must have been called.
             Each input of the composite operator must be redirected
             exactly once.
             Each input of all sub-operators must be the target of
             exactly one redirection.
*/
```

```
  void redirectOutput(int outputDS, APT_Operator* sub, int subOutputDS);
  /*
     effect    Specifies that the indicated output of this composite
               operator is to be tied to the indicated sub-operator's
               indicated output.
     note      The redirection is performed immediately.
               The composite operator's output adapter is moved onto
               the indicated sub-operator.
     requires  0 <= outputDS < outputDataSets()
               0 <= subOutputDS < sub->outputDataSets()
               setOutputDataSets() must have been called.
               sub non-null
               markSubOperator(sub) must have been called.
               Each output of the composite operator must be redirected
               exactly once.
               Each output of all sub-operators must be the target of
               exactly one redirection.
  */

  void setSubOpsSeparatelyConstrained();
  /*
     effect    By default, the composite operator's availableNodes()
               are propagated to the sub-operators.
               This member function can be called by the composite
               operator's describeOperator() function to turn off this
               action.  This would be appropriate if the
               describeOperator() routine applied separate constraints
               to each sub-operator.
     note      The default propagation of availableNodes() is *not*
               also performed for nodeMap()-- this is always the
               responsibility of the composite's describeOperator()
               logic [no good reason for this, but we can't compatibly
               change the behavior].
  */
};

#endif // APT_COMPOSITE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_CONFIG_H
#define APT_CONFIG_H

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_LOCATOR_H
#include <apt_util/locator.h>
#endif

class APT_PropertyList;
class APT_Archive;
class APT_NodeSet;
class APT_Node;
class APT_NodeResource;
class APT_RMSystem;

class APT_NodeSetRep;
class APT_NodeRep;
class APT_NodeResourceRep;

class APT_Config
/* Note: an APT_Config object is immutable.  Its state is determined
   by the configuration file; the client can observe, but not modify, its
   state.  So think of APT_Config as a read-only view of the config.apt
   file.
*/
{
public:
  static const APT_Config& get(bool dontInitialize=false);
  /* singleton instance; its state is determined by the configuration file */

private:
  friend class APT_Config_UT;
  APT_Config(const char* configFilePath=0, bool dontInitialize=false);
  /*
     effect    Constructs this config object by reading the indicated
               config file.  If no configFilePath argument is supplied,
               then the configuration file is looked for as follows.
```

```
              If the configFilePath does not lead to a valid
              ORCHESTRATE configuration file or, configFilePath is 0,
              and one can not be found with the rules below, then
              isValid() will be false.
    rules     First, if the user explicitly specified where to find
              the config file by setting $APT_CONFIG_FILE, look only
              there.  Second, try "./config.apt".  Third, try
              $APT_ORCHHOME/etc/config.apt.
    note      Use the APT_Config::get() function to access the
              APT_Config object corresponding to the usual config
              file.
  */

public:
  APT_Locator configFilePath() const;
  /*
    effect    Returns the file system path of the config file.
  */

  void initializeFrom(const APT_String& configFileContent);
  /*
    effect    Initializes this config from the indicated copy of a
              config file's content, replacing this config's previous
              state.
    note      If isValid() was false, it becomes true.
  */

  void initializeFrom(const APT_String& configFileContent,
                      const APT_String& configFileName);
  /*
    effect    Initializes this config from the indicated copy of a
              config file's content, replacing this config's previous
              state.  Sets the stored name of the config file to
              configFileName.
    note      If isValid() was false, it becomes true.
  */

  APT_String initializedContent() const { return configFileContent_; }
  /*
    effect    Returns non-empty string if initializeFrom() was
              called, returning the argument that was passed to
              initializeFrom().
  */

  bool isValid() const;
  /*
    effect    Tells if this config was successfully initialized.
  */

  APT_NodeSet allNodes() const;
  /*
    effect    Returns all nodes of this configuration.
```

```
   requires  isValid() true.
*/

APT_NodeSet nodesInPool(const char* nodePool) const;
/*
   effect    Returns the set of nodes in the indicated node pool.
             Returns the empty set if the argument string does not
             name any node pool.
   note      An empty nodePool selects for nodes in the default node
             pool.
             Node pool names are case-insensitive.
   requires  Arg non-null.
             isValid() true.
*/

APT_NodeSet nodesWithResourceKind(const char* resKind) const;
APT_NodeSet nodesWithResourceName(const char* resName) const;
APT_NodeSet nodesWithResourcePool(const char* resPool) const;
APT_NodeSet nodesWithResourceKindInPool
            (const char* resKind, const char* resPool) const;
/*
   effect    Returns the set of nodes with a resource, where the
             resource is specified by kind, name, or pool.
   note      An empty resPool selects for resources in the default
             pool for their resource kind.
             Resource names are case-sensitive; resource kinds and
             pools are case-insensitive.
   requires  Args non-null.
             isValid() true.
*/

bool hasNodesInPool(const char* nodePool) const;
/*
   effect    Returns whether the set of nodes in the indicated node
             pool is not empty.
   note      An empty nodePool selects for nodes in the default node
             pool.
             Node pool names are case-insensitive.
   requires  Arg non-null.
             isValid() true.
*/

bool hasNodesWithResourceKind(const char* resKind) const;
bool hasNodesWithResourceName(const char* resName) const;
bool hasNodesWithResourcePool(const char* resPool) const;
bool hasNodesWithResourceKindInPool
     (const char* resKind, const char* resPool) const;
/*
   effect    Returns whether the set of nodes with a resource is
             not empty, where the
             resource is specified by kind, name, or pool.
   note      An empty resPool selects for resources in the default
```

```
                pool for their resource kind.
                Resource names are case-sensitive; resource kinds and
                pools are case-insensitive.
     requires   Args non-null.
                isValid() true.
  */

  APT_NodeSet nodesWithVirtualMemory(APT_Int32 size) const;
  APT_NodeSet nodesWithPhysicalMemory(APT_Int32 size) const;
  /*
     effect     Returns the set of nodes with at least the indicated
                amount of virtual or physical memory.
     note       Size is in units of megabytes (2**20 bytes)
     requires   isValid() true.
  */

  APT_NodeSet equivalentNodes(const APT_Node&) const;
  APT_NodeSet equivalentNodes(const char *) const;
  /*
     effect     Returns the set of nodes that are equivalent to this
                node for communications purposes.  Specifically, the
                set of nodes with the identical fastname.
     requires   Arg non-null
  */

  static APT_Node myNode();
  /*
     effect     Returns the node object corresponding to the calling
                operator partition.
     requires   This function may only be called from within an operator's
                runLocally() function.
                isValid() true.
  */

  static APT_Node mainNode();
  /*
     effect     Returns the first node in the configuration that is
                     physically located on the physical host on which the
                sequential program is running.
     note       APT_Config::mainNode().nodeName() != "" is
                     guaranteed
     requires   isValid() true.
  */

  ~APT_Config();

  // TBD: add persistence

private:
  // prohibit copy/assign
  APT_Config(const APT_Config&);
  APT_Config& operator= (const APT_Config&);
```

```
  void completeInitialization(APT_RMSystem*);

  APT_Locator configFilePath_;
  APT_String configFileContent_;
  APT_NodeSet* allNodes_;
};

class APT_NodeSet
/* For all set operations that test node equality, the
   APT_Node::nodeName() is compared (case-insensitively) to determine
   node equality.
*/
{
public:
  APT_NodeSet();
  /*
     effect    Makes an empty node set.
  */
  APT_NodeSet(const APT_Node&);
  /*
     effect    Makes a node set with a single entry, the supplied node.
  */

  ~APT_NodeSet();
  APT_NodeSet(const APT_NodeSet&);
  APT_NodeSet& operator= (const APT_NodeSet&);

  enum LookupPath
  {
    eByNodeNameThenFastName=0,
    eByFastNameThenNodeName,
    eByNodeName,
    eByFastName                 /* finds *first* node with given fastname */
  };

  bool hasNode(const char*, LookupPath p=eByNodeNameThenFastName) const;
  /*
     effect    Tells if there is a node with the given name in this
               set.
               Name comparison is case-insensitive.
     requires  Arg non-null.
  */
  APT_Node lookup(const char*, LookupPath p=eByNodeNameThenFastName) const;
  /*
     effect    Returns the node with the given name.
               Name comparison is case-insensitive.
               Returns a default-constructed APT_Node if no node
               matched the given name.
     requires  Arg non-null.
  */
```

```
bool contains(const APT_Node&) const;
/*
   effect    Tells if the indicated node is a member of this node
             set.
   note      This is based on APT_Node::nodeName() comparison.
*/

bool isEmpty() const;
/*
   effect    Tells if this node set is empty.
*/

int numNodes() const;
/*
   effect    Tells how many nodes are in this set.
*/

APT_Node node(int index) const;
/*
   effect    Returns the indicated node of this set.
   note      Nodes are indexed in config file order.
   requires  0 <= index < numNodes()
*/

bool operator== (const APT_NodeSet&) const;
bool operator!= (const APT_NodeSet& rhs) const { return !(*this == rhs); }
/*
   effect    Compares two node sets based on whether they have the
             same APT_Node elements (as determined by comparing
             APT_Node::nodeName()s).
   note      Set comparison is independent of the ordering of their
             elements.
*/

APT_NodeSet computeUnion(const APT_NodeSet&) const;
/*
   effect    Returns the union of this node set and the argument
             node set.
   note      I wanted to name this function union(), but 'union' is
             a keyword of the C/C++ language.
*/

APT_NodeSet computeIntersection(const APT_NodeSet&) const;
/*
   effect    Returns the intersection of this node set and the
             argument node set.
*/

APT_NodeSet computeComplement(const APT_NodeSet& space) const;
/*
   effect    Returns the complement of this node set with respect to
             the argument node set.
```

```
   requires  All members of this node set must be members of the
             argument node set.
*/

// mutators

void addNode(const APT_Node&);
/*
   effect    A copy of the indicated node is added to this node
             set.  If the indicated node is default constructed
             or already a member, there is no effect.
   note      Adding nodes in config file order is most efficient.
*/

void removeNode(const APT_Node&);
/*
   effect    The indicated node is removed from this node set.
             This function does nothing if the indicated node is
             default constructed or is not a member of this set.
   note      Removing nodes in reverse config file order is most
             efficient.  For example, to remove all nodes:

             while (nodes.numNodes()>0)
               nodes.removeNode(nodes.node(nodes.numNodes()-1));
*/

void unionWith(const APT_NodeSet&);
/*
   effect    Modifies this node set to become the union of it and
             the argument node set.
*/

void intersectWith(const APT_NodeSet&);
/*
   effect    Modifies this node set to become the intersection of it
             and the argument node set.
*/

void addSet(const APT_NodeSet& s) { unionWith(s); }
/*
   effect    Each of the argument's nodes are addNode()ed to this
             node set.
   note      This is the same as unionWith().
*/

void removeSet(const APT_NodeSet&);
/*
   effect    Each of the argument's nodes are removeNode()ed from
             this node set.
*/

void unparse(ostream&, int indent=0) const;
```

```
  APT_String unparse(int indent=0) const;
  /*    Prints or returns a semi-human-readable list of the nodes.
        Used (for now) for debugging.
   */

  // TBD: add persistence

private:
  APT_NodeSetRep* rep_;
};

class APT_Node
/* Note: an APT_Node object is immutable, except for assignment.  All
   APT_Node instances are obtained (directly or indirectly) from the
   APT_Config object.
*/
{
public:
  APT_Node();
  /*
     effect    Constructs with empty names, empty property list, and
               no resources.
     note      A default-constructed APT_Node is useless as anything
               other than being the destination of an assignment.
  */
  ~APT_Node();
  APT_Node(const APT_Node&);
  APT_Node& operator= (const APT_Node&);

  bool operator== (const APT_Node&) const;
  /*
     effect    Returns true if the two nodes have the same nodeName().
               Node names are compared case-insensitively.
  */

  bool isEquivalent(const APT_Node&) const;
  /*
     effect    Returns true if the two nodes have the same fastName().
               Node names are compared case-insensitively.
  */

  APT_String nodeName() const;
  /*
     effect    Returns the name of this node.
     note      Node names should be compared case-insensitively.
               Within a given configuration, all node names are
               unique.
  */

  APT_String fastName() const;
  /*
     effect    Returns the "fast" name of this node.  By default, a
```

```
              node's fastName() is the same as its nodeName(), but a
              different fast name can be specified in the config file.
   note       Fast names should be compared case-insensitively.
              Within a given configuration, all fastnames are
              unique.
              Except where a node's fastname is the same as its
              node name, no fastname will be the same as any node name.
*/

bool isInPool(const char* nodePool) const;
/*
   effect     Tells if this node is a member of the given node pool.
   note       If nodePool is the empty string, this is interpreted as
              the "default node pool".
              Node pool comparison is case-insensitive.
   requires   Arg non-null.
*/

int numPools() const;
/*
   effect     Returns the number of node pools to which this node
              belongs.
*/

APT_String pool(int index) const;
/*
   effect     Returns the name of one of the node pools to which this
              node belongs.
   requires   0 <= index < numPools()
*/

APT_Int32 virtualMemory() const;
APT_Int32 physicalMemory() const;
/*
   effect     Returns the amount of virtual or physical memory on
              this node.
   note       Return value is in units of megabytes (2**20 bytes)
*/

int numResources() const;
/*
   effect     Returns the number of resources attached to this node.
*/

APT_NodeResource resource(int index) const;
/*
   effect     Returns a resource attached to this node.
   requires   0 <= index < numResources()
*/

int numResources(const char* resKind) const;
/*
```

```
     effect    Returns the number of resources of the given kind
               attached to this node.
               Resource kind comparison is case-insensitive.
     requires  resKind non-null
  */

  APT_NodeResource resource(const char* resKind, int index) const;
  /*
     effect    Returns a resource of the given kind attached to this
               node.
               Resource kind comparison is case-insensitive.
     requires  resKind non-null
               0 <= index < numResources(resKind)
  */

  int numResources(const char* resKind, const char* resPool) const;
  /*
     effect    Returns the number of resources of the given kind
               in the given pool.
               Resource kind and pool comparison is case-insensitive.
     requires  resKind and resPool non-null
  */

  APT_NodeResource resource(const char* resKind, const char* resPool, int index)
const;
  /*
     effect    Returns a resource of the given kind in the given pool.
               Resource kind and pool comparison is case-insensitive.
     requires  resKind and resPool non-null
               0 <= index < numResources(resKind, resPool)
  */

  APT_PropertyList properties() const;
  /*
     effect    Returns the property list (if any) for this node.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_Node&);
  /* serialization needed for APT_Operator's nodeMap stuff; should not
     see general use. */

private:
  friend class APT_Config;
  friend class APT_NodeSet;
  APT_Node(APT_NodeRep*);       // copies rep object

  APT_NodeRep* rep_;
};

class APT_NodeResource
/* Note: an APT_NodeResource object is immutable, except for
   assignment.  All APT_NodeResource instances are obtained (directly
```

```
   or indirectly) from the APT_Config object (via APT_Node instances).
*/
{
public:
  APT_NodeResource();
  /*
     effect    Constructs with empty kind, name, properties, pools.
     note      A default-constructed APT_NodeResource is useless as
               anything other than being the destination of an
               assignment.
  */
  ~APT_NodeResource();
  APT_NodeResource(const APT_NodeResource&);
  APT_NodeResource& operator= (const APT_NodeResource&);

  APT_String kind() const;
  /*
     effect    Returns what kind of resource this is; e.g., disk,
               scratchdisk, tape, etc.
  */

  APT_String name() const;
  /*
     effect    Returns the name of this resource.  This is typically
               of the form /dev/mumble, and is case-sensitive.
  */

//  bool exclusiveUse() const;
  /*
     effect    Tells whether this is an exclusive-use resource (true),
               or this is a shared resource (false).
  */

  bool isInPool(const char* resPool) const;
  /*
     effect    Tells if this resource is a member of the given
               resource pool.
               Resource pool comparison is case-insensitive.
     requires  Arg non-null.
  */

  int numPools() const;
  /*
     effect    Returns the number of resource pools to which this
               resource belongs.
  */

  APT_String pool(int index) const;
  /*
     effect    Returns the name of one of the resource pools to which this
               resource belongs.
     requires  0 <= index < numPools()
```

```
  */

  APT_PropertyList properties() const;
  /*
     effect    Returns the property list (if any) for this resource.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_NodeResource&);
  /* serialization needed for APT_Node serialization; should not
     see general use. */

private:
  friend class APT_Config;
  APT_NodeResourceRep* rep_;
};

#endif // APT_CONFIG_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_CURSOR_H
#define APT_CURSOR_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

class APT_OperatorRep;
class APT_DataSetRep;
class APT_FieldSelector;
class APT_Schema;
class APT_Archive;
class APT_InputAccessorBase;
class APT_InputTagAccessor;
class APT_InputSubCursor;
class APT_OutputAccessorBase;
class APT_OutputTagAccessor;
class APT_OutputSubCursor;

class APT_CursorBase
{
  APT_DECLARE_RTTI(APT_CursorBase);
protected:
  APT_CursorBase();             // abstract class

public:
  virtual ~APT_CursorBase();

  bool isSetup() const { return ds_ ? true : false; }
  /*
```

```
    effect    Tells if this cursor has been setup.
  */

  APT_Schema schema() const;
  /*
    effect    Returns the schema against which field accessors, tag
              accessors, and sub-cursors may be bound.
    note      If this cursor was setup from within an operator,
              the cursor's schema() is the interface schema of the
              operator's dataset.
              If this cursor was setup directly from a dataset,
              the cursor's schema() is the concrete schema of the
              dataset.
    requires  isSetup() is true.
  */

  friend APT_Archive& operator|| (APT_Archive& ar, APT_CursorBase&)
  { return ar; }
  /*
    effect    Cursor serialization is supported so that operators'
              cursor data members (if any; typically operators do not
              have cursor data members) can be serialized along with
              other operator data members.
              However, cursor serialization is a nop: the "setup"
              state of an cursor is not serialized.
  */

protected:
  friend class APT_DataSetRep;

  APT_DataSetRep* ds_;
  enum CursorType { eOperator, eDataSet };
  CursorType ctype_;

private:
  APT_CursorBase(const APT_CursorBase&);
  APT_CursorBase& operator= (const APT_CursorBase&);
};

class APT_InputCursor : public APT_CursorBase,
                        public APT_InputAccessorInterface
{
  APT_DECLARE_RTTI(APT_InputCursor);

public:
  APT_InputCursor();
```

```
  /*
     effect    Makes an uninitialized cursor, which is essentially
               unusable.  The APT_Operator and APT_DataSet classes
               have an setupInputCursor() member function that
               initializes an input cursor that may then be used to
               obtain access to records.
  */
  ~APT_InputCursor();
  // note: cursor must be destroyed before data set to which it is bound

  // copy/assign prohibited in base class

  // resolve ambiguous functions
  bool isSetup() const { return APT_CursorBase::isSetup(); }
  APT_Schema schema() const { return APT_CursorBase::schema(); }

  // accessor setup functions in APT_InputAccessorInterface

//  void setLookaheadWindow(int numRecords);
//  int lookaheadWindow() const;
// pre-context, post-context?
// window shrinks on EOF
  /*
     effect    Controls the number of input records that are available
               to be accessed via setLookahead().  The default is 1.
     note      The input lookahead window should be sized to fit
               comfortably into memory.  The implementation makes no
               attempt to efficiently utilize secondary storage to
               manage a huge lookahead window.
     requires  numRecords >= 1
  */

//  bool setLookahead(int index);
// or setPositionInWindow()?  also relative motion?
  /*
     effect    Specifies an index identifying which record will be
               read by accessor objects.
               An index of 0 means the input record about to be
               discarded by getRecord().  Successive index values
               correspond to successive input records.
               Defaults to 0.
     note      Whenever getRecord() is called, the input lookahead is
               reset to 0.
               Any sub-cursors bound to this cursor are reset
               to their 0th positions.
     returns   true: there is an input record at the indicated
```

```
                position in the lookahead window.
              false: there is no input record at the indicated
                position in the lookahead window (EOF was reached).
                In this case, the lookahead() index is not modified.
    requires  0 <= index < lookaheadWindow()
              isSetup() is true.
  */

//  int lookahead() const;
  /*
    effect    Returns the current lookahead index for this cursor.
    note      Whenever getRecord() is called, the input lookahead is
              reset to 0.
    requires  isSetup() is true.
  */

  bool getRecord(int skip=0);
  /*
    effect    Advances the cursor to the next input record.  The
              previously current input record is discarded; if an
              input lookahead window has been established, the
              "oldest" input record is discarded.
              The lookahead() becomes 0.
              Returns true if there is an input record; returns false
              if there is no input record.
              The skip argument, if non-zero, specifies how many
              records are to be skipped while advancing the cursor.
    note      This must be called at least once before accessing
              input record data.
              Any sub-cursors bound to this cursor are reset
              to their 0th positions.
    requires  0 <= skip
              isSetup() is true.
  */

  // EXPERIMENTAL
  void currentRecordBuffer(void* *addr, APT_UInt32* size);
  /*
    effect    Retrieves the location and length of the current
              record.
              Presumably the caller will copy the record buffer away
              for later submission to setCurrentFromBuffer().
    note      The returned information is invalidated by the next
              call to getRecord().
              The storage format of the record data is unspecified
              and subject to change from one Orchestrate release to
```

```
                the next.
     requires   getRecord() must have been called, and the most recent
                call must have returned true.
  */
  // EXPERIMENTAL
  void setCurrentRecordFromBuffer(const void* addr, APT_UInt32 size);
  /*
     effect     Sets this input cursor's current record.
     requires   The addr and size arguments must refer to a record
                buffer that was obtained via currentRecordBuffer().
                If the record buffer has been copied, then addr's
                alignment (mod 8) must be the same as the addr value
                returned from currentRecordBuffer().
  */

  // EXPERIMENTAL
  void setEOF();
  // TBW

  // EXPERIMENTAL
  void abandon();
  /*
     effect     Abandons any remaining input for this cursor. All
                subsequent calls to getRecord() will return false.

     requires   isSetup() is true.

     note       This is currently implemented as if getRecord(INT_MAX)
                is called repeatedly until EOF. A better implementation
                will be more intelligent; see isAbandoned(), below.
  */

private:
  friend class APT_DataSetRep;

  void didSetup();

  friend class APT_CombinableOperator;
};

class APT_OutputCursor : public APT_CursorBase,
                         public APT_OutputAccessorInterface

{
  APT_DECLARE_RTTI(APT_OutputCursor);
```

```
public:
  APT_OutputCursor();
  // copy/assign prohibited in base class

  ~APT_OutputCursor();
  // note: cursor must be destroyed before data set to which it is bound

  void putRecord();

  // EXPERIMENTAL
  bool isAbandoned() const;
  /*
     effect    Returns true if all consumers of the virtual dataset
               associated with this cursor have called abandon() on
               their input cursors. Subsequent calls to putRecord()
               will be ignored.

     note      This currently returns false unconditionally.
  */

  void putRecordAndFlushPartition();
  /*
     effect    Like putRecord(); in addition causes any buffered
               records to be sent on their way.
     note      Only the partition(s) being written by the current
               record is flushed.
  */

  void done();
  /*
     effect    Flushes and closes this output cursor.  No further
               putRecord() operations may be performed after a done().
  */

  // resolve ambiguous functions
  bool isSetup() const { return APT_CursorBase::isSetup(); }
  APT_Schema schema() const { return APT_CursorBase::schema(); }

private:
  friend class APT_DataSetRep;

  void didSetup();
};

#endif // APT_CURSOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_DATASET_H
#define APT_DATASET_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_ISDYNAMIC_H
#include <apt_util/isdynamic.h>
#endif

#ifndef APT_PROPLIST_H
#include <apt_util/proplist.h>
#endif

#ifndef APT_OPERATOR_H
#include <apt_framework/operator.h> /* for APT_Operator's PartitionMethod
                                       and CollectionMethod */
#endif

class APT_Int53;
class APT_Schema;
class APT_InputCursor;
class APT_OutputCursor;
class APT_DataSetRep;
class APT_DataSetSC;
class APT_DMDataSet;
class APT_Partitioner;
class APT_Collector;
class APT_ViewAdapter;
class APT_GeneralSubprocessOperator;
class APT_Record;

class APT_DataSet : public APT_Persistent, public APT_IsDynamic
{
  APT_DECLARE_RTTI(APT_DataSet);
  APT_DECLARE_PERSISTENT(APT_DataSet);

public:
  ~APT_DataSet();

  APT_DataSet(const char* ident=0);
  /*
```

```
   effect    Constructs a virtual dataset.  A virtual dataset is
             used to connect two operators and flow records in one
             direction between them.
*/
APT_DataSet(APT_Operator* writer, int writerDS,
            APT_Operator* reader, int readerDS,
            const char* ident=0);
/*
   effect    Constructs a virtual dataset and attaches it to the
             indicated ports of the indicated operators.
   requires  writer non-null.
             reader non-null.
             See APT_Operator::attachInput() and
             APT_Operator::attachOutput() for additional
             requirements.
*/
bool isVirtual() const;
/*
   effect    Tells if this is a virtual dataset.
*/

enum Direction { eInput, eOutput };
enum OutputFlag
{
  eCreate,
  eReplace,
  eAppend,
  eDiscardRecords,
  eDiscardSchemaAndRecords,   // file-based datasets only
  eKeepSchema=eDiscardRecords, // synonym
  eKeepPartitioning=eDiscardSchemaAndRecords // synonym
};
enum DSType {eDataSet, eSasDataSet, eSeqSasDataSet};
APT_DataSet(const char* filename, Direction dir, OutputFlag f=eCreate,
            const char* ident=0, DSType type=eDataSet);
/*
  effect         Constructs a file-based persistent dataset for either
                 input or output operation.
                 An input persistent dataset must exist and contain
                 schema information.
                 The OutputFlag is ignored for input persistent
                 datasets.
                 If the output persistent dataset does not already
                 exist, it will be created (see the note below).  A
                 newly created APT_DataSet's hasSchema() will
                 initially be false.
                 If the OutputFlag is eDiscardRecords or
                 eDiscardSchemaAndRecords, a warning is issued if
                 the dataset does not already exist.
                 If an output persistent dataset already exists, the
                 OutputFlag determines this constructor's behavior:
```

```
                  * eCreate (default): an error is reported.
                  * eReplace: the existing dataset is first removed
                    and an entirely new dataset created in its place.
                  * eAppend: the existing dataset's attributes
                    (including schema) are retained, and any records
                    written to it will be appended to the dataset as
                    a new segment.  hasSchema() and canSetSchema()
                    will be true and false, respectively, if the
                    dataset exists already and therefore has a
                    schema.
                    If the data set does not exist, it is created.
                  * eDiscardRecords: the existing dataset's
                    attributes (including schema) are retained, but
                    any existing records are discarded by removing
                    all segments.  New records are added to a newly
                    appended segment in the specified disk pool.
                    hasSchema() and canSetSchema() will be true and
                    false, respectively, if the dataset exists
                    already and therefore has a schema.
                    If the data set does not exist, it is created.
                  * eDiscardSchemaAndRecords: some of the existing
                    dataset's attributes are retained, but its schema
                    and any existing records are discarded (as in
                    eDiscardRecords).
                    hasSchema() and canSetSchema() will be false and
                    true, respectively.
                    If the data set does not exist, it is created.
  note            Any difficulty constructing this file-based persistent
                  dataset can be discerned via the errorLog() object.
                  The files for an output dataset may not actually be
                  created until run time (for datasets that are part
                  of a step) or when an output cursor is setup (for
                  datasets directly written from conductor code).
  requires        filename non-null
*/

DSType type() const;
/*
   effect    Tells us the actual dataset type.
*/

void setType(DSType);
/*
   effect    Sets dataset type.
*/

bool isFile() const;
/*
   effect    Tells if this is a file-based persistent dataset.
*/
APT_String filename() const;
```

```
/*
   effect    Returns the filename with which this file-based
             persistent dataset was constructed.
   requires  isFile() is true.
*/
OutputFlag outputFlag() const;
/*
   effect    Returns the OutputFlag argument with which this file-based
             persistent dataset was constructed.
   requires  isFile() is true.
*/

bool hasError() const;
/*
   effect    Indicates whether this persistent dataset (file-based)
             has encountered an error during its initialization.
*/
APT_String getError() const;
/*
   effect    If hasError() is true, returns an explanatory string.
             Returns an empty string otherwise.
*/

APT_ErrorLog& errorLog();
/*
   effect    Returns access to this data set's internal error log
             object.
   note      Any traffic generated on this log will be identified as
             originating from this data set.
*/

/* generic operations (these apply to any kind of dataset) */

bool isInput() const;
/*
   effect    Tells if this dataset is a source of records to be read
             by an operator.
             True for virtual datasets and file-based persistent
             datasets initialized for input.
*/

bool isOutput() const;
/*
   effect    Tells if this dataset is a destination for records to
             be written by an operator.
             True for virtual datasets and file-based persistent
             datasets initialized for output.
*/

APT_Operator* consumingOperator();
const APT_Operator* consumingOperator() const;
```

```
/*
   effect    Returns the consuming operator to which this data
             set is attached.
             Returns 0 if this data set has not been attached to
             a consuming operator.
   requires  isInput() must be true.
*/

APT_Operator* producingOperator();
const APT_Operator* producingOperator() const;
/*
   effect    Returns the producing operator to which this data
             set is attached.
             Returns 0 if this data set has not been attached to
             a producing operator.
   requires  isOutput() must be true.
*/

bool hasSchema() const;
/*
   effect    Tells if this dataset has a schema.
*/

APT_Schema schema() const;
/*
   effect    Returns a copy of this dataset's schema.
   requires  hasSchema() is true.
   TBD       When does a virtual dataset's (or output persistent
             dataset's) schema become known?  Probably after
             step-check.
*/

APT_Node* nodeMap(int* nodeMapLength,
                  APT_ErrorLog * log = 0) const;
/*
   effect   Generates a node map that can be used to read the
            dataset without any repartitioning. The number
            of items is stored in nodeMapLength.

            If a node map cannot be generated for the dataset,
            null is returned and nodeMapLength is set to 0.
            If an error log has been provided, it may contain
            appropriate messages.

            The caller is responsible for releasing the storage
            used by the map (via "delete [] nodeMap");

   requires isFile() == true
            isInput() == true
            hasError() == false
*/
```

```
bool canSetSchema() const;
/*
   effect    Tells if it is legal to call setSchema().  This
             function returns false if:
             - this is a virtual dataset and schema has already
               been propgated to it, or
             - this is an input persistent dataset, or
             - this is an output persistent dataset constructed as
               eAppend or eDiscardRecords and the existing dataset
               (if any) already has a schema, or
             - a cursor has been setup for this dataset, or
             - the step containing this dataset has been checked.
*/

void setSchema(const APT_Schema& sch);
/*
   effect    Sets this dataset's schema.  The argument is copied.
   requires  canSetSchema() is true.
             sch.isConcreteSchema() must be true.
             The argument must not have a constructError().
*/

void setDiskPool(const char*);
/*
   effect    Specifies the set of disks on which an output file data
             set is created.  By default, data files are put on the
             disks in the default pool.  The name of the default pool
             is the zero length string ("").
   note      The disk pool doesn't influence the number of partitions
             or node placement of the dataset or its producing
             operator.  Rather, the output file dataset is created
             with the same number of partitions and node placement as
             its producing operator.  The disk pool only determines
             which disks the data files will be placed on when the
             dataset is created by the framework.  That is why each
             node on which the producing operator runs must have at
             least one disk in the specified disk pool (or default
             disk pool if this function is not called).
   requires  arg non-null.
             isOutput() must be true.
             isFile() must be true.
             Each node on which the producing operator runs must have
             at least one disk in the disk pool.
*/

void setPartitionMethod(APT_Operator::PartitionMethod m);
/*
   effect    Specifies the partitioning method to be used on this
             data set.
```

```
            By default, the partition method is determined by the
            operator whose input this data set is feeding.
   note     If this is an output persistent data set, the framework
            inserts a repartition operator to accommodate this
            data set's partitioner.
   requires m must not be APT_Operator::eAny nor
            APT_Operator::eOther.
            This data set must be attached to an operator (cannot
            set partition method on data set to be directly
            accessed).
            If this data set is attached to an operator's input
            port, that operator must be parallel, and the
            operator's partition method for that input port must be
            eAny.
            This function should not be called more than once per
            data set object.
*/
void setPartitionMethod(APT_Partitioner* part,
                        const APT_ViewAdapter& adapter);
/*
   effect   Specifies the partitioner object to be used on this
            data set.
            The adapter argument is used to adapt this data set's
            concrete schema to the partitioner's input interface
            schema.
            The adapter object is copied.
            By default, the partition method is determined by the
            operator whose input this data set is feeding.
   note     If this is an output persistent data set, the framework
            inserts a repartition operator to accommodate this
            data set's partitioner.
   requires part non-null
            part should be dynamically allocated.  It will be
            deleted by the framework.
            This data set must be attached to an operator (cannot
            set partition method on data set to be directly
            accessed).
            If this data set is attached to an operator's input
            port, that operator must be parallel, and the
            operator's partition method for that input port must be
            eAny.
            This function should not be called more than once per
            data set object.
*/

bool hasPartitionMethod() const;
/*
   effect   Returns true if either form of setPartitionMethod has been
            called for this dataset, otherwise false.
*/
```

```
APT_Operator::PartitionMethod partitionMethod() const;
/*
   effect    Returns the partition method to be used for this dataset.
   requires  hasPartitionMethod() == true
*/

const APT_Partitioner* partitioner() const;
/*
   effect    Returns a pointer to the partitioner to be used by this
             dataset if one was specified via setPartitionMethod; otherwise
             returns null.
*/

APT_Partitioner* removePartitioner();
/*
   effect    Returns a pointer to the direct partitioner to be used by this
             dataset if one was specified via setPartitionMethod; otherwise
             returns null. In the case of a keyless partitioner (eRoundRobin,
              for example) NULL is also returned. The partitioner is removed
             and setPartitionMethod may be called again. The caller owns
             the partitioner object and is responsible for deleteing it if
             necessary (subject to isDynamic()).
   requires  The consuming operator's describeOperator() must not yet have
             completed when this function is called.
*/

void setCollectionMethod(APT_Operator::CollectionMethod m);
void setCollectionMethod(APT_Collector* coll,
                         const APT_ViewAdapter& adapter);
/* Analogous to the setPartitionMethod() functions, except for feeding
   a sequential operator */

bool hasCollectionMethod() const;
/*
   effect    Returns true if either form of setCollectionMethod has been
             called for this dataset, otherwise false.
*/

APT_Operator::CollectionMethod collectionMethod() const;
/*
   effect    Returns the collection method to be used for this dataset.
   requires  hasCollectionMethod() == true
*/

const APT_Collector* collector() const;
/*
   effect    Returns a pointer to the collector to be used by this
             dataset if one was specified via setCollectionMethod; otherwise
             returns null.
*/
```

```
APT_Collector* removeCollector();
/*
   effect    Returns a pointer to the direct collector to be used by this
             dataset if one was specified via setCollectionMethod; otherwise
             returns null. In the case of a keyless collector (eRoundRobin,
              for example) NULL is also returned. The collector is removed
             and setCollectionMethod may be called again. The caller owns
             the collector object and is responsible for deleteing it if
             necessary (subject to isDynamic()).
   requires  The consuming operator's describeOperator() must not yet have
             completed when this function is called.
*/

bool hasPreservePartitioningFlag() const;
/*
   effect    Tells if this dataset's preserve partitioning (PP) flag
             value has been determined.
   note      An input persistent dataset's PP flag is always
             available.  Other kinds of dataset's PP flag is not
             available until either step check occurs, or the
             dataset's PP flag is explicitly set or cleared.
*/

bool preservePartitioningFlag() const;
/*
   effect    Returns the preserve partitioning (PP) flag for this
             dataset.
             Returns true if the PP flag is set and false if the PP
             flag is clear.
   requires  hasPreservePartitioningFlag() must be true.
*/

void setPreservePartitioningFlag();
void clearPreservePartitioningFlag();
/*
   effect    Sets or clears the preserve partitioning (PP) flag for
             this dataset.
   note      This action overrides the PP flag that would be
             propagated by the framework or set by an operator (on
             its output dataset(s)).
             If this is an input persistent dataset, its PP flag is
             overridden just for this APT_DataSet instance; the
             PP flag stored in the persistent dataset is not
             modified.
*/


bool hasCombiningOverrideFlag() const;
/*
   effect    Tells if this dataset's combining override flag value
             has been set.
```

```
*/

bool combiningOverrideFlag() const;
/*
   effect    Returns the combining override flag for this dataset.
             Returns true if setCombiningOverrideFlag() was called
             and false if clearCombiningOverrideFlag() was called.
   requires  hasCombiningOverrideFlag() must be true.
*/

void setCombiningOverrideFlag();
void clearCombiningOverrideFlag();
/*
   effect    Sets or clears the combining override flag for this
             dataset.
             Both calls set the hasCombiningOverrideFlag() flag.
*/


void setupInputCursor(APT_InputCursor* cur);
void setupOutputCursor(APT_OutputCursor* cur);
/*
   effect    Initializes the given cursor object so it can be used
             to read (write) field data from (to) this dataset.
   note      The cursor's schema() is the concrete schema of this
             dataset.
   requires  cur non-null.
             cur->isSetup() should be false upon entry.
             This dataset must not already have a cursor.
             hasError() must be false.
             This dataset must not be attached to an operator.
             This must be a persistent dataset.
             isInput() (isOutput()) must be true for setupInputCursor()
             (setupOutputCursor()).
             This dataset must have a schema.
             The dataset's schema must not be changed after any cursors
             are initialized.
             For output datasets, the outputFlag() must be eCreate or
             eReplace.
             For output datasets, the calling program must be the
             application's main program; the caller cannot be
             executing from an operator's runLocally().
   OBSOLETE
*/

APT_Int53 recordCount() const;
/* requires

     `isFile () && isInput ()'.  ie., it is a
     file-based persistent dataset open for reading.
```

```
     hasError() must be false.

   returns
     the number of records in the persistent input dataset.
     If the dataset has had setFirstSegment and/or setLastSegment called
     on it, the number returned will be the number of segments in the
     range specified by the first and last segments specified.
 */

APT_String ident() const;
/*
   effect    Returns a string by which this data set may be
             identified.
             If setIdent() has not been called (or has been called
             with an empty string arg), returns:
                persistent data set: the file name
               virtual data set: the empty string
*/
void setIdent(const char*);
/*
   effect    Determines what ident() returns.
             This is used by OSL to set a data set's ident() to be
             the data set's name as it appears on the command line.
             If the data set is an unnamed virtual data set, OSL
             qualifies its name like foo_op(o1) (indicating the
             virtual data set is written by output 1 of foo_op).
   requires  Arg non null.
*/

void remove();
/*
   effect    Removes this persistent dataset completely from
             persistent storage, including any contents and
             persistent description.
   note      This dataset object should not be used after a call to
             remove().
   requires  isVirtual() false.
             hasError() false.
*/

int numberOfSegments() const;
/*
  effect     Returns the number of append segments in the dataset.
             If the dataset is open for writing, the current
             append segment is not included.  Segments with no
             records on any or all partitions are counted.  The
             number of segments is equal in all partitions.
  requires   isFile() (file-based persistent dataset).
             hasError() must be false.
*/
```

```
bool willReadAllSegments() const;
void setWillReadAllSegments(bool);
/*
   effect    If true, causes all segments of this data set to be
             read for each step segment of a checkpointed step.
             The default is false.
   requires  Must be a file-based input persistent data set.
*/


/**** Buffer control ****/

enum BufferingPolicy
{
  eInheritBuffering,   /* use the buffering policy inherited from the step */
  eAutomaticBuffering, /* buffer if necessary to prevent dataflow deadlock */
  eForceBuffering,     /* unconditionally buffer this virtual dataset */
  eNoBuffering         /* do not buffer this virtual dataset */
};

void setBufferingPolicy(BufferingPolicy bp);
BufferingPolicy bufferingPolicy() const;
/*
   effect    Sets/gets the buffering policy for this dataset.
             Buffering takes memory, CPU time and disk space;
             but it may improve dataflow and prevent deadlock.
             Changing the buffering policy of a dataset has no
             influence on the buffering of other virtual datasets.
             Initially, the buffering policy is eInheritBuffering.
   note      Setting the buffering policy of a persistent dataset
             usually has no effect because a persistent dataset is
             inherently buffered.  But if it is repartitioned, its
             buffering policy applies to the automatically inserted
             virtual dataset.
   warning   Specifying eNoBuffering on a virtual dataset could
             cause a dataflow deadlock if done inappropriately.
*/

void setBufferingParameters(const APT_PropertyList& pl);
APT_PropertyList bufferingParameters() const;
/*
   effect    Sets/gets the buffering parameters for this dataset.
             These parameters only affect a virtual dataset that is
             actually buffered.  The property list is initially empty.
   note      Default buffering parameters for the whole step can be
             set by calling APT_Step::setBufferingParameters().
   note      Setting the buffering parameters of a persistent dataset
             usually has no effect.  But if it is repartitioned, its
             buffering parameters apply to the automatically inserted
             virtual dataset if it is buffered.
   warning   Setting the buffering parameters inappropriately could
```

```
              cause a dataflow deadlock during step execution.

    properties (all are optional):

    { maximumMemoryBufferSize=size   // (*) Maximum memory used for buffering
      bufferFreeRun=size             // (*) How much buffering until the
                                     // buffer operator offers resistance
                                     // to the flow of data
                                     // Note: the size should be slightly less
                                     // than 2/3 of maximumMemoryBufferSize.
                                     // The amount less will control how
                                     // much data is flushed to disk in
                                     // one operation.
      maximumTimeout=time            // (*) Not currently documented.
      initialTimeout=time            // (*) Not currently documented.
      baseQueueTimeout=time          // (*) Not currently documented.
      outputTurnoverRate=time        // (*) Not currently documented.
      outputAdjustmentThreshold=size // (*) Not currently documented.
      minimumDeltaT=time             // (*) Not currently documented.
      reportThreshold=size           // (*) Not currently documented.
      inputLowWaterMark=size         // (*) Not currently documented.
      timeout2scale=size             // (*) Not currently documented.
    } // (*) starred items are optional
  */


  /**** Voodoo ****/

  APT_Record* concreteInputRecord();
  /*
     effect    Returns the concrete record object representing the current
               input record on this data set.  This is intended to support
               operators that need to bind multiple input interfaces to
               the same input data set.
     requires  This data set must be operating in input mode (that is,
               an operator may call this function on an input data set,
               but not an output data set).
  */


private:
  friend class APT_SC; // access to scInterface()
  friend class APT_GeneralSubprocessOperator;
  friend class APT_OperatorRep;
  friend class APT_DataSetSC;
  friend class APT_DataSet_UT;  // for testing

  // prohibit copying
  APT_DataSet(const APT_DataSet&);
  APT_DataSet& operator= (const APT_DataSet&);
```

```
  const APT_DataSetSC* scInterface() const;
  APT_DataSetSC* scInterface();
  /*
     effect    Returns a pointer to an object providing supplemental
               interface to this dataset.  For the step compiler
               implementation.
     note      The caller must not delete the returned pointer.
  */

  APT_DMDataSet* dmObj();
  /*
     effect    Returns a pointer to the internal "data manager" data
               set object for this APT_DataSet.
  */

  void markRecordLoopInstance(bool tf);
  /*
     effect    Tell this instance whether it will be used for the main
               record loop.  Default is true.
  */
  void markTearDownInstance(bool tf);
  /*
     effect    Tell this instance whether it has responsibility for
               tearing down.  Default is true.
  */

public:                          // bogus...
  APT_DataSetRep* rep_;
};

#endif // APT_DATASET_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_FIELDSEL_H
#define APT_FIELDSEL_H

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_Archive;
class APT_String;
class APT_Lexer;
class APT_ParseError;

class APT_FieldSelector
/* A schema field selector is the name of a component within a schema.
   To name a sub-component, a dot-delimited list of component
   identifiers is given.  To name a vector component, subscripting may
   be used.

   For example, consider the following schema:

   record
   (
     a: int;
     b[]: subrec
     (
       x: int;
       y[]: sfloat;
     );
   )

   The following field selectors identify components of the above
   schema:

   a            The "a" field
```

```
   b[1].x       The x sub-component of the second instance of the b[]
                vector
   b[0].y[2]    The third element of the y[] vector subcomponent of the
                first instance of the b[] vector

   The APT_FieldSelector class encapsulates field selectors.  In
   addition to providing access to the components and subscripts (if
   any) of a field selector, the class provides parse/unparse
   operations implementing a string-based representation of valid
   field selectors.
*/
{
public:
  APT_FieldSelector() : numComponents_(0), components_(0) {}
  /*
     effect    Initializes numComponents() to 0.
  */

  APT_FieldSelector(const APT_FieldSelector&);
  APT_FieldSelector& operator= (const APT_FieldSelector&);

  ~APT_FieldSelector()
  { if (numComponents_ == 1) delete components_;
    else if (numComponents_) delete[] components_;
  }

  APT_FieldSelector(const char*, APT_ParseError* err=0);
  /*
     effect    Parses the string, expecting a field selector.
               If the parse is successful, this APT_FieldSelector
               object's state is replaced by the parsed field
               selector.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 field selector.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
     requires  string arg non-null
  */

  int numComponents() const { return numComponents_; }
  /*
     effect    Tells how many dot-delimited components this field
               selector has.
  */
  void setNumComponents(int);
  /*
     effect    Adjusts the number of components.  If growing,
               the right-most components are initialized to
               empty identifiers; if shrinking, the right-most
```

```
            components are discarded.
*/

int addComponent(const char* id, int subscript=-1, int pos=-1);
int addComponentId(const APT_Identifier& id, int subscript=-1, int pos=-1);
/*
   effect    Adds a component to this field selector.
             The subscript argument indicates that the added
             component should have a subscript specification with
             the given value; a subscript value of -1 means the
             added component has no subscript.
             The pos argument specifies what position the
             component is to occupy; a pos value of -1 means add
             the component to the end.
             Components are numbered left-to-right.  Component 0
             is the left-most component.
   returns   The position index at which the component was added.
   requires  0 <= pos < numComponents()+1  or pos==-1
             id non-null
             id must contain only letters, digits, and underscore,
               and must not begin with a digit.
             0 <= subscript  or subscript==-1
*/

void removeComponent(int pos);
/*
   effect    Removes the component at the indicated position.
   requires  0 <= pos < numComponents()
*/

const APT_Identifier& componentName(int pos) const;
void setComponentName(int pos, const APT_Identifier& id);
/*
   effect    Accesses the indicated component's identifier.
   requires  0 <= pos < numComponents()
             id non-null
             id must contain only letters, digits, and underscore,
               and must not begin with a digit.
*/

int componentSubscript(int pos) const;
void setComponentSubscript(int pos, int subscript);
/*
   effect    Accesses the indicated component's subscript.  A
             subscript value of -1 indicates that the component has
             no subscript.
   requires  0 <= pos < numComponents()
             0 <= subscript  or subscript==-1
*/
```

```
  bool isSubscripted() const;
  /*
     effect    Tells if any component of this field selector is
               subscripted.
  */

  void parse(const char*, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a field selector.
               If the parse is successful, this APT_FieldSelector
               object's state is replaced by the parsed field
               selector.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 field selector.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
               If an error occurs, this APT_FieldSelector object's
               state is unchanged.
     requires  string arg non-null
  */

  void unparse(ostream&) const;
  APT_String unparse() const;
  /*
     effect    Prints or returns a parse()able representation of
               this APT_FieldSelector.
  */

  friend bool operator== (const APT_FieldSelector&,
                          const APT_FieldSelector&);
  /*
     effect    Tells if the two field selectors are identical in every
               observable way.
     note      Component name comparision is case-independent.
  */
  friend bool operator!= (const APT_FieldSelector& lhs,
                          const APT_FieldSelector& rhs)
  { return !(lhs == rhs); }

  APT_UInt32 hash() const;

  friend APT_Archive& operator|| (APT_Archive&, APT_FieldSelector&);

  void repInvariant() const;

  void parse_(APT_Lexer&, APT_ParseError* err, bool embedded=false);

private:
```

```
  struct Component
  {
    APT_Identifier id;
    int subscript;              // -1 means none

    Component() : subscript(-1) {}

    APT_DECLARE_NEW_AND_DELETE(Component);
  };

  int numComponents_;
  Component* components_;
};
APT_DIRECTIONAL_SERIALIZATION(APT_FieldSelector);

#endif // APT_FIELDSEL_H
```

```
// -*-Mode: C++-*-
//
// Copyright (C) 1996 Torrent Systems, Inc. All Rights Reserved.
//
// $Id: fifocon.h,v 1.12 1998/07/31 20:41:45 smr Exp $

#ifndef APT_FIFOCON_H
#define APT_FIFOCON_H

#ifndef APT_GSUBPROC_H
#include <apt_framework/gsubproc.h>
#endif

class APT_String;

class APT_FifoConnectionImpl;
class APT_FifoConnection: public APT_GeneralSubprocessConnection
{
  public:

    APT_FifoConnection();
    ~APT_FifoConnection();

    APT_FifoConnection(Direction direction,
                       const char * envVarName);
    /*
       effects  Create a fifo-based connection with the specified
                direction. The name of the fifo will be generated,
                and that name will be exported to an environment
                variable with the specified name.
    */


    virtual void setFifoName(const char * name);
    virtual void setEnvVarName(const char * name);
    /*
       effects  Set the name of the fifo or the environment
                variable that will contain the fifo name.

       requires name must not be null.

                For setFifoName, if name exists, it must be the name of
                a fifo, and not in use.

       notes    If setFifoName is not called, a name will be generated.

                If setFifoName is called and the fifo does not exist,
                it will be created, used, and deleted at the end of
                the step. If it exists, it will be used and NOT deleted.
    */

    virtual APT_String identifier() const;
    /*
       effects  Returns "file [<fifoname>,] $<envvarname>"
```

```
    */

    // void setDirection(Direction direction) is inherited.

    virtual APT_String fifoName() const;
    virtual APT_String envVarName() const;
    virtual int localFileDescriptor() const;
    /*
       effects  Accessors for the name of the fifo, the
                name of the environment variable containing
                the name of the fifo, and the file descriptor
                used to access the local end of the fifo.

       requires These may only be called from runConnection().
    */

  protected:

    virtual APT_Status
      reportConnectionAttachment(const APT_GeneralSubprocessConnection * connection);
    virtual APT_Status create(const APT_GeneralSubprocessOperator * op);
    virtual APT_Status disableConnection();
    virtual APT_Status enableConnection();
    virtual APT_Status setupInSubprocess(APT_GeneralSubprocessOperator * op);
    virtual APT_Status terminateConnection();
    virtual bool supportsFileSets() const;
    virtual bool supportsDirectFiles() const;

    virtual void setupBuffers();

    APT_FifoConnectionImpl * pImpl_;

  private:

    APT_DECLARE_RTTI(APT_FifoConnection);
    APT_DECLARE_PERSISTENT(APT_FifoConnection);
};

#endif // APT_FIFOCON_H
```

```
// -*-Mode: C++-*-
//
// Copyright (C) 1996 Torrent Systems, Inc. All Rights Reserved.
//
// $Id: gsubproc.h,v 1.27 2000/02/17 16:20:09 smr Exp $

#ifndef APT_GSUBPROC_H
#define APT_GSUBPROC_H

#ifndef APT_OPERATOR_H
#include <apt_framework/operator.h>
#endif

#ifndef APT_DATASET_H
#include <apt_framework/dataset.h>
#endif

class APT_String;
class APT_GeneralSubprocessOperatorImpl;
class APT_GeneralSubprocessConnection;
class APT_BufferInput;
class APT_BufferOutput;
class APT_FileSet;
class APT_SubprocConnection;

class APT_GeneralSubprocessOperator : public APT_Operator
{
  public:

    APT_GeneralSubprocessOperator();
    virtual ~APT_GeneralSubprocessOperator();
    // Copy contruction and assignment prohibited by base class

    enum Destination {
        eStdout,
        eStderr
      };

  protected:

    // Step check time function (called from describeOperator())

    APT_UInt32
      addConnection(const APT_GeneralSubprocessConnection & connection);
    /*
       effects  Attach the specified connection to this operator. A copy
                of the connection will made and associated with the
                operator. The id number of the connection is returned.

                At step execution time a process will be created for
                the connection and runConnection() will be called
                with the id number and a pointer to the connection.

                setId() will be called on the connection after it is
```

```
            cloned.

   notes    All subprocesses receive, by default:

            - fd 0 (stdin) open to /dev/null
            - fd 1 (stdout) set up to write to stdout of the main program
            - fd 2 (stderr) set up to write to stderr of the main program

            These may be overridden by connections attached to the
            operator.

            The id numbers returned by addConnection are ascending
            integers starting with zero. The programmer may exploit
            this to avoid having to save the number, but be careful:
            calling redirectConnection consumes a position.
*/


void
  redirectConnection(const APT_GeneralSubprocessConnection & connection,
                     Destination destination,
                     APT_UInt32 lrecl = 0);
/*
   effects  Redirect output from the specified connection to either
            stdout or stderr. If lrecl is zero, the output is presumed
            to be variable length, newline-delimited text; if lrecl is
            non-zero, data is take from the connection in lrecl sized
            chunks and each chunk is sent as a line.

   requires connection must be a sink (output) connection.
*/


APT_UInt32
  addFileSetConnection(const APT_GeneralSubprocessConnection & connection,
                       const APT_FileSet & fileSet,
                       bool overwrite=false, bool append=false,
                       bool noCleanup=false);
/*
   effects  Attach the specified connection to this operator, and
            associate it with the specified fileset. This causes
            files in the file set to be directly connected to
            the subprocess via the connection, with no subprocess
            created.

            A fileset defines a nodemap; this map must applied to the
            operator in describeOperator, it is not done automatically.
   requires connection.supportsFileSets() must be true
*/

APT_UInt32
  addDirectFileConnection(const APT_GeneralSubprocessConnection & connection,
                          const char* filename,
                          bool overwrite=false, bool append=false,
                          bool noCleanup=false);
```

```
  /*
     effects  Attach the specified connection to this operator, and
              associate it with the specified file.  This causes
              the file to be directly connected to the subprocess
              via the connection, with no subprocess created.

     note     This operator player must be co-located with the file;
              this is the responsibility of describeOperator() (or
              the OSL SIL transformation that led to this being a
              direct file connection).
     requires connection.supportsDirectFiles() must be true
  */

  APT_UInt32
    addSubprocConnection(const APT_SubprocConnection& connection);
  /*
     effects  Attach the connection to this operator.  This causes the
              operator to attach directly to a named pipe (that has another
              direct subproc attachement on it other end).  This connection
              is purely passive in so far as we set it up, bind it to appropriate
              fd's or fifo's on the subproc ports, and then get the heck out
              of the way.  As such we don't need to create a connection process
              for a subproc connection.
  */

  APT_SubprocConnection* subprocConnection(APT_UInt32 connectionId);
  /*
     effect   Retrieve a subprocess connection
  */


  void remapSubprocConnectionPort(APT_DataSet::Direction dir,
                                  int dataSetNumber,
                                  APT_UInt32 connectionId);
  /*
     effect   Remaps the subproc connection port to bind it to the
              correct dataset identifier.  Checks to make sure
              that each dataSetNumber is assigned at most once.
   */



  // Step execution time functions

  virtual APT_Status describeConnectionPorts()=0;
  /*
     effect   This function will be called by runLocally() after
              the create() function has been called for each of
              the connections attached to the operator. Success
              or failure status is returned.
     note     This is called just before describeCommandLine().
     requires This function must call setConnectionPort() for all
              data set ports before returning.
  */
```

```
   void setConnectionPort(APT_DataSet::Direction, int dataSetNumber,
                          APT_UInt32 connectionId);
   /*
      effect   Declares which connection object is responsible for
               reading or writing each data set port.
      requires Each data set port for this operator must have exactly
               one connection object associated with it.
               Each connection object may be associated with zero or
               one data set ports.
   */

   virtual APT_Status describeCommandLine()=0;
   /*
      effects  This function will be called by runLocally() after
               the create() function has been called for each of
               the connections attached to the operator. Success
               or failure status is returned.

      requires This function must call either setCommandLine() or
               setCommandArgs() before returning.
   */


   void setCommandLine(const char * commandLine);
   void setCommandArgs(int argc, const char * argv[]);
   /*
      effects  Specify the arguments to be used to invoke the
               subprocess. If setCommandLine() is called, the
               specified character string is broken into tokens
               at whitespace boundaries (tabs, spaces, newlines);
               the first token is used as the command name, and
               subsequent tokens are passed as arguments. If
               setCommandArgs() is called, argc is the number of
               pointers in argv; the first entry in argv points
               to the name of the command, and subsequent entries
               are used as arguments.

      requires One, and only one, of these functions must be called
               by describeCommandLine().

               For setCommandLine():

               - commandLine must not be null.

               - There must be at least one token in the string.

               For setCommandArgs():

               - argc must be at least 1 (for the command name).

               - None of the entries in argv may be null.
   */


   virtual APT_Status
```

```
      runConnection(APT_GeneralSubprocessConnection * connection)=0;
    /*
       effects  This routine is called in each process that functions
                as a driver for a connection. The id of the connection
                being driven by this process and a pointer to the
                connection object are supplied. The routine decides
                which connection it is handling and performs the
                appropriate processing.  Success or failure status is
                returned.
    */


    virtual APT_Status setupInSubprocess();
    /*
       effects  This is called immediately prior to the exec() that
                starts the subprocess command running. It provides one
                last opportunity to alter the environment of the
                subprocess. Success or failure status is returned.

                The default implementation does nothing other than
                return APT_StatusOk.
    */


    virtual APT_Status checkTermination(int status);
    /*
       effects  This is called after all of the connection processes
                and the actual subprocess have terminated. It is passed
                the subprocess termination status value (from waitpid);
                it should decide if this represents normal or abnormal
                termination and return sucess or failure. This also
                provides one last cleanup point.
                The default implementation returns APT_StatusOk if the
                termination status was zero, APT_StatusFailed otherwise.
       note     This function is not called if a TerminationChecker
                callback has been provided.
    */

public:
    virtual APT_Status freeUpFileDescriptor(int fd);
    /*
       effects This is called by any connection that needs to take
               ownership of a given file descriptor.  The subproc
               will call all of its connections and ask that they
               free the descriptor
    */

public:
    class TerminationChecker : public APT_Persistent
    {
      APT_DECLARE_RTTI(APT_GeneralSubprocessOperator::TerminationChecker);

APT_DECLARE_ABSTRACT_PERSISTENT(APT_GeneralSubprocessOperator::TerminationChecker);

    public:
```

```
     virtual APT_Status terminationCallback(int status)=0;
     // works like checkTermination()
   };

   void setTerminationChecker(TerminationChecker*);
   /*
      effect    Alternative to the checkTermination() function.
                If a TerminationChecker callback object is provided,
                then checkTermination() is not called.
                A null argument resets the callback.
      requires  The argument must be dynamically allocated.
      note      The argument object becomes owned by this operator.
   */

protected:

   APT_UInt32 connectionCount() const;
   const APT_GeneralSubprocessConnection * connection(APT_UInt32 id) const;
   APT_GeneralSubprocessConnection * connectionPtr(APT_UInt32 id);
   /*
      effects  Accessors for the number of connections and for the
               connections themselves.

      requires id < connectionCount
   */

   void setEnvironmentVariable(const char * name,
                               const char * value) const;
   /*
      effects  This can be called from setupInSubprocess() to
               set an environment variable for the subprocess.
               If the variable is already set, the old value is
               discarded and replaced with the new value.

      requires name, value must not be null.

               This may only be called from setupInSubprocess.
   */

 private:

   APT_Status checkTermination_(int status);
   /* calls termination callback, if any. Otherwise, calls
      checkTermination(). */

   APT_Status runLocally();    // not to be overridden further

   APT_GeneralSubprocessOperatorImpl * pImpl_;
   TerminationChecker* callback_;

   friend class APT_GeneralSubprocessOperatorImpl;
   friend class APT_GeneralSubprocessConnection;
   friend class APT_GeneralSubprocessConnectionImpl;

   APT_DECLARE_RTTI(APT_GeneralSubprocessOperator);
```

```
    APT_DECLARE_ABSTRACT_PERSISTENT(APT_GeneralSubprocessOperator);
};


class APT_GeneralSubprocessConnectionImpl;
class APT_GeneralSubprocessConnection : public APT_Persistent
{
  public:

    // Whatever the class needs to define the connection: file descriptor
    // numbers for the subprocess side, names of environment variables to
    // be set to FIFO names, etc.


    enum Direction {
        eUnspecified,           // not yet known
        eSource,                // provides data to the subprocess
        eSink                   // consumes data from the subprocess
      };


    // These functions provide a buffered transport layer.  These calls
    // are implemented in terms of the inputBuffer() and outputBuffer()
    // objects (see below)

    const char * obtainInput(APT_Int32 request, APT_Int32 *actual);
    /*
       effect   Requests that the indicated amount of input be made
                available.

       returns  Non-null: input is available; the amount of input
                 available is written to the actual arg.  Actual will
                 be less than request if the end of input has been
                 reached before the entire request could be
                 satisfied.
                Null: No input is available; 0 will be written to the
                 actual arg.

       requires The consumedInput() function must be called before
                obtainInput() can be called again.

                request > 0

                It must be appropriate to receive on this connection.
    */


    void consumedInput(APT_Int32 amount);
    /*
       effect   Indicates how much of the data from obtainInput() has
                been consumed by the caller.

       requires obtainInput() must have been called.

                Must be called once before calling obtainInput() again.
```

```
            amount <= actual of previous obtainInput() call.

            It must be appropriate to receive on this connection.
*/


char * prepareOutputBuffer(APT_Int32 request);
/*
   effect   Requests that buffer space be made available for the
            indicated number of bytes.

   requires The wroteOutput() function must be called before
            prepareOutputBuffer() can be called again.

            request > 0

            It must be appropriate to send on this connection.
*/


void wroteOutput(APT_Int32 amount);
/*
   effect   Indicates how much of the buffer from
            prepareOutputBuffer() has been written by the caller.

   requires prepareOutputBuffer() must have been called.

            Must be called once before calling
            prepareOutputBuffer() again.

            amount <= request of previous prepareOutputBuffer() call.

            It must be appropriate to send on this connection.
*/


void flushOutput();
/*
   effect   Forces all buffered output to be written.

   requires The most recent prepareOutputBuffer() call must have
            had a corresponding wroteOutput() call.

            It must be appropriate to send on this connection.
*/


Direction direction() const;
/*
   effects  Returns the direction of the connection.
*/


APT_UInt32 id() const;
```

```
   /*
      effects  Returns the id of the connection.
   */


   void setDirection(Direction direction);
   /*
      effects  Set the direction of the connection. This will
               typically be set in the constructor.
   */


   APT_BufferInput *  inputBuffer();
   APT_BufferOutput * outputBuffer();
   /*
      effects  Provide access to the internal input or output buffer
               associated with the connection.

      requires For outputBuffer, direction() == eSource.
               For inputBuffer,  direction() == eSink.
   */

   void setInterruptable(bool state);
   bool getInterruptable() const;
   /*
      effects  A connection is considered to be a "reliable" component
               that will NOT hang indefinitely as a result of actions
               take by the subprocess. Since an output connection may
               wind up blocked by downstream flow control, connections
               are killed by SIGUSR1 while in runConnection(). SIGUSR1
               is, by default, IGNORED by runConnection() code. This
               allows connections to drain.

               A connection, though, may KNOW that is about to take
               action that could cause an indefinite hang. In this
               case, it should call setInterruptable(true) to allow
               itself to be killed by SIGUSR1; when the action
               is complete, setInterruptable(false) should then be
               called to disable unwanted termination.
   */


   virtual bool supportsFileSets() const;
   /*
      returns  True if the connection supports filesets; else false.
               The default implementation returns false.
   */

   virtual bool supportsDirectFiles() const;
   /*
      returns  True if the connection supports direct file access; else false.
               The default implementation returns false.
      note     This is generally the same as supportsFileSets().
   */
```

```
  bool hasFileSet() const;
  const APT_FileSet & getFileSet() const;
  /*
     effects  Returns a reference to the fileset attached via setFileSet.

     requires hasFileSet true.
              supportsFileSets true.
  */

  bool hasDirectFile() const;
  APT_String getDirectFile() const;
  /*
     effects  Returns the name of the file specified via setDirectFile().

     requires hasDirectFile() true.
              supportsDirectFiles() true.
  */


  virtual APT_String identifier() const;
  /*
     effects  Returns a string that can be used in error message to
              identify the connection in a manner that makes sense
              to the end user. The direction of the connection should
              *not* be part of the identifier.

              The default implementation returns "connection <n>".
  */

protected:

  APT_GeneralSubprocessConnection();
  virtual ~APT_GeneralSubprocessConnection();
  // copy construction and assignment prohibited.


  void
    setFileSet(const APT_FileSet & fileSet, bool overwrite, bool append,
               bool noCleanup);
  /*
     effects  Use the specified fileset to source or sink the
              connection, instead of a subprocess.

     requires The connection must support use of a fileset.
  */

  void
    resetFileSet(const APT_FileSet & fileSet, bool overwrite, bool append,
                 bool noCleanup);
  // re-assign existing file set

  void
    setDirectFile(const char* filename, bool overwrite, bool append,
                  bool noCleanup);
```

```
   /*
      effects  Use the file to source or sink the connection, instead
               of a subprocess.

      requires The connection must support use of a direct file.
   */

   bool hasOverwrite() const;
   bool hasAppend() const;
   bool hasNoCleanup() const;
   // requires: fileset or direct file only

   virtual APT_Status
     reportConnectionAttachment(const APT_GeneralSubprocessConnection * connection);
   /*
      effects  This routine is called by addConnection to allow existing
               connections to check that the new connection does not
               conflict with them. A pointer to the new connection
               is supplied; success or failure status is returned.

               Typically a checked cast (using APT_PTR_CAST) will be
               used to determine if the new connection is of the same
               type as this connection; if so, checking will be done.

               The default implementation simply returns APT_StatusOk.
   */


   void setId(APT_UInt32 id);
   /*
      effects  Set the id of the connection. setId() will be called
               by addConnection().
   */

   virtual APT_Status create(const APT_GeneralSubprocessOperator * op)=0;
   /*
      effects  This is called by runLocally() of the instance of
               APT_GeneralSubprocessOperator that this connection is
               attached to, prior to creation of any of the subprocesses
               that implement the operator. The connection should do
               whatever is necessary to create the connection between
               the source/sink process and the subprocess (such as create
               a pipe, FIFO, shared memory region or whatever). If the
               connection is successfully created, APT_StatusOk must be
               returned; otherwise APT_StatusFailed must be returned.

               The "op" argument is a pointer to the operator that will
               be using the connection. The connection can ignore this,
               but may want to save it for use in error messages.
   */


   virtual APT_Status disableConnection()=0;
   /*
      effects  This is called in processes that will serve as the
```

```
            "Orchestrate end" of OTHER connections, not this one.
            The connection should take steps to insure that data is
            not inadvertently transferred between this process and
            the target subprocess (by, for instance, closing the
            pipe, detaching a shared memory region, etc.). Success
            or failure is returned to the caller.
   */


   virtual APT_Status enableConnection()=0;
   /*
      effects  This is called in the process that will serve as the
               "Orchestrate end" of THIS connections. The connection
               can do anything necessary to complete the connection
               to the target subprocess. Success or failure is
               returned to the caller.
   */


   virtual APT_Status setupInSubprocess(APT_GeneralSubprocessOperator* op)=0;
   /*
      effects  This is called in the process that will ultimately become
               the target subprocess (following exec()). It can, for example,
               establish environment variables, dup() file descriptors to
               appropriate numbers, etc. Success or failure is returned to
               the caller.
   */


   virtual APT_Status terminateConnection();
   /*
      effects  This is called in processes that will served as the
               driver for this connection after runConnection() has
               return. The connection should release any resources
               associated with the connection and return a success
               or failure status.

               The default implementation simply returns APT_StatusOk.
   */


   void setEnvironmentVariable(const char * name,
                               const char * value) const;
   /*
      effects  This can be called from setupInSubprocess() to
               set an environment variable for the subprocess.
               If the variable is already set, the old value is
               discarded and replaced with the new value.

      requires name, value must not be null.

               This may only be called from setupInSubprocess.
   */
```

```
    // Accessors for use by derivations.

    bool
      isRedirected() const;

    APT_GeneralSubprocessOperator::Destination
      destination() const;

    APT_UInt32
      lrecl() const;

    void setInputBuffer(APT_BufferInput*); // must be sink
    void setOutputBuffer(APT_BufferOutput*); // must be source
    // arg becomes owned by the base connection object

    int
      partitionNumber() const;

  private:

    // Copy construction and assignment prohibited.
    APT_GeneralSubprocessConnection(const APT_GeneralSubprocessConnection & rhs);
    APT_GeneralSubprocessConnection &
      operator = (const APT_GeneralSubprocessConnection & rhs);

    friend class APT_GeneralSubprocessOperator;
    friend class APT_GeneralSubprocessOperatorImpl;
    friend class APT_OSL_SubprocOperator;

    // These functions are called by APT_GeneralSubprocessOperator
    // and should not be overridden; that's why they are private.

    APT_GeneralSubprocessConnection * clone() const;

    virtual void setupBuffers()=0;
    // must be overridden to call setInputBuffer() or setOutputBuffer()

    APT_Status redirect();

    APT_GeneralSubprocessConnectionImpl * pImpl_;

    APT_DECLARE_RTTI(APT_GeneralSubprocessConnection);
    APT_DECLARE_ABSTRACT_PERSISTENT(APT_GeneralSubprocessConnection);
};


#endif // APT_GSUBPROC_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_IMPEXP_FUNCTION_H
#define APT_IMPEXP_FUNCTION_H

#ifndef APT_FUNCTION_H
#include <apt_framework/type/function.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

class APT_SchemaField;
class APT_ErrorLog;

class APT_GFImportExport : public APT_GenericFunction
{
protected:
  APT_GFImportExport(const char* schemaTypeName,
                     const char* implementationName);
  // passes args through to base ctor; interfaceName is "import_export"

  // default copy/dtor OK; assign prohibited in base class.
  // copy is only for purposes of clone() method.

public:

  /* Parameterization interface.  Import/export generic functions are
     parameterized as follows:

     0. setDirection() is called.
     1. setArgType() is called (inherited virtual function).
     2. setIsNullable() is called.
     3. setParams() is called, perhaps multiple times, to convey
        format parameters.  If called multiple times, the call order
        is record-level, followed by subrec level(s), if any, followed
        by field-level parameters.  The intention is that later calls'
        format directives override format information from earlier
        calls.
     4. setLengthMode() is called.
```

```
    5. setDropped() or setGenerated() is called, if appropriate.
    6. validateParameters() is called at check-time, and should be
       used to detect/report parameterization errors.
    7. cacheParameters() is called at run-time, and can be used to
       pre-compute information needed by the run-time functions.

    Note: serialization is performed between validateParameters() and
    cacheParameters().
  */

  enum Direction { eImport=0, eExport };

  Direction direction() const { return direction_; }
  void setDirection(Direction d) { direction_ = d; }
  /*
     effect    Accesses the translation direction of this field.
               Defaults to eImport.
     note      This information is serialized in the base class.
  */

  enum LengthMode
  {
    eNone=0,            /* no framework-handled length (either fixed-
                           length, or this import/export function
                           handles the length as part of the field's
                           external representation) */
    ePrefixed,          // field group driver handles length using prefix
    eLinked,            // ditto, using linked field
    eDelimited          // ditto, using trailing delimiter (or quotes)
  };

  LengthMode lengthMode() const { return lengthMode_; }
  void setLengthMode(LengthMode lm) { lengthMode_ = lm; }
  /*
     effect    Accesses the length mode of this field.
               Defaults to eNone.
     note      This information is serialized in the base class.
  */

// reminder...
//  virtual APT_GenericFunction* clone() const=0;
  /* effect    See base class description.
     requires   Specific for this GF, you may not call clone on an
                     object that has had setGenerated() called on it.
  */

  bool isNullable() const { return isNullable_; }
  void setIsNullable(bool nullable) { isNullable_ = nullable; }
  /*
     effect    Tells if the field being imported or exported is
               nullable.
```

```
    note      Most derivations can ignore this attribute; some
              import functions want to conditionalize their
              behavior depending on whether the result is
              nullable.
  */

//  virtual void setParams(const APT_PropertyList& pl,
//                         APT_PropertyList* unrecognized);
  /*
    effect    See base class description.
    note      Even if one or more properties are unrecognized,
              overrides of this function must process all valid
              properties, while shunting unwanted properties to
              *unrecognized.
              This function can be called multiple times-- at
              record-level, at each subrec level, and on the field
              itself.
              Overrides must arrange for a later call's properties to
              overwrite an earlier call's properties.
              Member data computed by this function should be
              serialized by the derived class.
  */

  bool isDropped() const { return isDropped_; }
  void setDropped(bool d) { isDropped_ = d; }
  /*
    effect    Indicates whether this field is being dropped when
              imported.  This might allow import() to be more
              efficient in this case.
              Default is false.
    note      This information is serialized in the base class.
  */

  const void* generated() const { return genVal_; }
  void setGenerated(const void* genVal) { genVal_ = genVal; }
  /*
    effect    Indicates whether this field is being generated when
              exported.  This might allow Export() to be more
              efficient in this case.
              If non-null, points to the constant value used to
              generate the external representation.
              Default is null.
    note      This information is serialized in the base class (well
              not really, but the generated() value, if any, will be
              available for cacheParameters()).
    requires  The supplied genVal must persist for the lifetime of
              this function instance.
  */

  virtual APT_Status validateParameters(APT_ErrorLog& log);
  /*
```

```
   effect    Should be overridden to check parameters, if any, for
             validity.
             The base implementation returns APT_StatusOk.
   return    APT_StatusOk: parameters are fine; warnings can be
               written to *log, if provided.
             APT_StatusFailed: problem with one or more parameters;
               *log should contain information describing
               problem(s).
   note      Member data computed by this function should be
             serialized by the derived class.
*/

virtual void cacheParameters();
/*
   effect    Can be overridden to compute and cache information (in
             member variables of the derived class) to facilitate
             speedy execution of the import() or Export() function.
             The base implementation is a nop.
   note      This function will not be called if the field group
             driver uses a direct copy instead of calling import()
             or Export().
*/


/* Field group driver interface.  These functions are called after
   validateParameters(), but before cacheParameters().
*/

virtual bool isFixedLength() const;
/*
   effect    Should be overridden if this type can have a
             fixed-length external representation.
             Base implementation returns false.
   note      The value returned by this function determines whether
             Export() or exportFixed() will be called, and whether
             import() or importFixed() will be called.
             setLengthMode() is called before isFixedLength().
   returns   true: as parameterized, this type has fixed-length
               external representation.
             false: as parameterized, this type has variable-length
               external representation.
*/
virtual APT_UInt32 fixedLength() const;
/*
   effect    If isFixedLength() can return true, this function
             should be overridden to return this type's fixed-length
             external representation.
             Base implementation calls APT_DETAIL_FATAL().
   requires  isFixedLength() true.
             Must not return zero.
*/
```

```
virtual bool isBoundedLength() const;
/*
   effect    Should be overridden if this type can have an external
             representation whose length has a bounded upper limit.
             Base implementation returns false.
   note      If isFixedLength() is true, then the caller shouldn't
             care whether isBoundedLength() is true or false.
   returns   true: as parameterized, this type has bounded-length
               external representation.
             false: as parameterized, this type has variable-length
               external representation with no strict upper limit.
*/
virtual APT_UInt32 boundedLength() const;
/*
   effect    If isBoundedLength() can return true, this function
             should be overridden to return this type's upper limit
             to the length of its external representation.
             Base implementation calls APT_DETAIL_FATAL().
   note      If isFixedLength() is true, then the caller should use
             fixedLength() in preference to boundedLength().
   requires  isBoundedLength() true.
             Must not return zero.
*/

virtual bool canCopy() const;
/*
   effect    Can be overridden to indicate whether this type's
             external representation is bitwise equal to its
             internal Orchestrate representation, AND that the
             prefix or linked field, if selected by lengthMode(),
             are to represent the field external length.
             Base implementation returns false.
   returns   true: as parameterized, this type's external
               representation is the same as its internal
               Orchestrate representation, AND the prefix/linked
               field (if any) is to be the field's external length.
             false: as parameterized, this type's external
               representation is different than its internal
               Orchestrate representation, OR the prefix/linked
               field (if any) is something other than the field's
               external length.
   note      The import/export group driver will not call this
             function if isFixedLength() is false.
*/

virtual bool handlesNulls() const;
/*
   effect    Can be overridden to indicate if this impexp generic
             function performs null recognition (import) and
             creation (export) in a type-specific manner, replacing
```

```
            field group driver mechanisms such as the null_field
            and null_length properties.
            Base implementation returns false.
   note     For eImport, if this function returns true but the
            field is not nullable, the field group driver issues a
            warning at check time.
*/

virtual APT_PropertyList explicitParams() const=0;
/*
   effect   Should be overridden to return a setParams() compatible
            property list representing the configured state of this
            generic function.
   note     The intention is that the returned property list be
            explicit about each of this gf's attributes; nothing
            should be defaulted.
            Also, the returned properties should be in some
            "canonical" order, to facilitate property list
            comparison.
            This function may not be called until after
            validateParams() has been called (and may be called
            either before or after cacheParams().
*/


/* Run-time interface.  These functions are called after
   cacheParamters() has been called.

   In the functions below, T refers to the field value type (C++
   built-in type or class) that the client sees when manipulating a
   field value via an accessor.
*/

enum Status { eOk=0, eBadField, eBadRecord, eShortInput, eNull };

virtual Status importFixed(const char* buffer, void* Tval, APT_UInt32 avail);
/*
   effect   Must be overridden to read from the buffer and write a
            value to Tval.
            The avail argument is the length of the available
            buffer contents.
            Base implementation calls APT_DETAIL_FATAL().
   returns  eOk: import operation successful.
            eBadField: import operation failed, but subsequent
              fields should not be affected by this field's import
              failure.  The field group driver substitutes a
              default value, if one has been specified.
            eNull: imported field should be set to null by the
              field group driver.  This is treated as eBadField
              by the field group driver if the field is not
              nullable.
```

```
            Note: importFixed() implementations are not expected to
            return eBadRecord or eShortInput.
   note     This function is called only when isFixedLength() is
            true.
            This function is not called if the field group driver
            has determined that it can import this field via a
            direct copy operation.
   guarantee The caller ensures that buffer points to fixedLength()
            bytes.
            The caller promises that the Tval value is "consumed"
            before the buffer contents are disturbed.  In
            particular, this allows things like
            APT_RawField::bind() to be used.
   requires  This function must not examine buffer beyond
            fixedLength() bytes.
*/

virtual Status import(const char* &buffer, APT_UInt32 len, APT_UInt32 avail,
                     bool endFlag, void* Tval, APT_ErrorLog* log);
/*
   effect    Must be overridden to read from the buffer and write a
            value to Tval.
            The len argument is supplied as follows, depending on
            lengthMode():
              eNone:      the length of buffer, to the end of the
                          current input record
              ePrefixed:  the value obtained from the prefix byte(s)
              eLinked:    the value obtained from the linked field
              eDelimited: the length of this field, as determined
                          by its delimiter (or quotes).
            The avail argument is the length of the available
            buffer contents.
            The endFlag argument is set to true iff the avail amount
            represents the entire input record.
            Base implementation calls APT_DETAIL_FATAL().
   returns   eOk: import operation successful.  The buffer pointer
              must have been incremented past this field's buffer
              representation (excluding any trailing delimiter,
              which is handled by the field group driver).
            eBadField: import operation failed, but subsequent
              fields should not be affected by this field's import
              failure.  The field group driver substitutes a
              default value, if one has been specified.
              The buffer pointer must have been incremented past
              this field's buffer representation (excluding any
              trailing delimiter, which is handled by the field
              group driver).
            eBadRecord: import operation failed, in such a way that
              this function could not determine the appropriate
              amount to advance the buffer pointer.  The group
              driver will likely reject the record (unless the
```

```
                current field has a default, and the following field
                has an absolute position directive; note this is not
                yet implemented-- the record is always rejected).
                If non-null, the log object should be used to communicate
                the reason eBadRecord is being returned (the error
                message should not have a trailing newline).
              eShortInput: The avail amount is insufficient for
                importing this field.
              eNull: imported field should be set to null by the
                field group driver.  This is treated as eBadField
                by the field group driver if the field is not
                nullable.
   guarantee  The caller ensures that buffer points to at least avail
              bytes.
              The caller promises that the Tval value is "consumed"
              before the buffer contents are disturbed.  In
              particular, this allows things like
              APT_RawField::bind() to be used.
   note       This function is called only when isFixedLength() is
              false.
   requires   The caller checks that the buffer pointer has not been
              incremented too far (beyond valid input data).
*/


virtual APT_UInt32 exportSize(const void* Tval);
/*
   effect     If isFixedLength() and isBoundedLength() can both be
              false, must be overridden to compute the amount of
              buffer space required by the supplied field value's
              external representation.
              If Tval is 0, then the size of a null export should be
              returned.  This will only occur if the field being
              exported is null and handlesNulls() is true.
              Base implementation calls APT_DETAIL_FATAL().
   note       An overestimate is OK.
              The import/export group driver will not call this
              function if either isFixedLength() or isBoundedLength()
              is true.
*/

virtual Status exportFixed(const void* Tval, char* buffer, APT_UInt32 avail);
/*
   effect     Must be overridden to write the given field value's
              external representation to the supplied buffer.
              If Tval is 0, then a null should be exported.  This will
              only occur if the field being exported is null and
              handlesNulls() is true.
              Base implementation calls APT_DETAIL_FATAL().
   returns    eOk: field export successful.
              eBadField: field could not be formatted to the given
```

```
                buffer.
              eBadRecord: formatting failure from which continuing
                is problematic (because the function detected that it
                overwrote the available buffer).
   note       This function is called only when isFixedLength() is
              true.
              This function is not called if the field group driver
              has determined that it can export this field via a
              direct copy operation.
   guarantee  The caller ensures that buffer points to fixedLength()
              bytes.
   requires   This function must write a value to all fixedLength()
              bytes pointed to by buffer, and not beyond.
*/

virtual Status Export(const void* Tval, char* &buffer,
                      APT_UInt32 avail, APT_UInt32* len,
                      APT_ErrorLog* log);
/*
   effect     Must be overridden to write the given field value's
              external representation to the supplied buffer.
              If Tval is 0, then a null should be exported.  This will
              only occur if the field being exported is null and
              handlesNulls() is true.
              The buffer argument must be incremented past this
              field value's external representation.
              The len return paramter should be written as follows,
              depending on lengthMode():
                eNone:      Don't care.
                ePrefixed:  Value to be encoded in this field's
                            prefix byte(s)
                eLinked:    Value to be back-stored in the linked
                            field
                eDelimited: Don't care. A delimiter (and/or quote
                            pair) is written (by the field group
                            driver) after this field's buffer
                            representation (as indicated by the
                            buffer pointer's final value).
              Base implementation calls APT_DETAIL_FATAL().
   returns    eOk: field export successful, buffer pointer advanced,
                and len written (if relevant).
              eBadField: field could not be formatted to the given
                buffer, buffer pointer advanced, and len written (if
                relevant).
              eBadRecord: field could not be formatted to the given
                buffer, buffer pointer not properly advanced, len not
                necessarily written.
                If non-null, the log object should be used to
                communicate the reason eBadRecord is being returned
                (the error message should not have a trailing newline).
   note       This function is called only when isFixedLength() is
```

```
                false.
     requires   buffer will point to adequate "avail" storage (as
                determined by exportSize(), fixedLength(), or
                boundedLength()).
                This function must write a value to all bytes between
                buffer's entry value and its exit value-1.
                This function must not write beyond the region
                determined by the avail argument.  The caller checks
                that the buffer pointer has not been incremented too
                far.
   */

protected:
  APT_GFImportExport(); // required for persistence

private:
  Direction direction_;
  LengthMode lengthMode_;
  bool isNullable_;
  bool isDropped_;
  const void* genVal_;

  APT_DECLARE_RTTI(APT_GFImportExport);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_GFImportExport);
};

#endif // APT_IMPEXP_FUNCTION_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_OPERATOR_H
#define APT_OPERATOR_H

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_ISDYNAMIC_H
#include <apt_util/isdynamic.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

class APT_DataSet;
class APT_Partitioner;
class APT_Collector;
class APT_ViewAdapter;
class APT_ModifyAdapter;
class APT_InputCursor;
class APT_OutputCursor;
class APT_TransferAdapter;
class APT_Schema;
class APT_FieldSelector;
class APT_NodeSet;
class APT_Node;
class APT_PropertyList;
class APT_ErrorLog;
class APT_String;

class APT_CompositeOperator;
class APT_OperatorRep;
class APT_OperatorSC;

class APT_Operator : public APT_Persistent, public APT_IsDynamic
/* Abstract base class defining protocol for all Orchestrate
   operators.
```

```
*/
{
  APT_DECLARE_RTTI(APT_Operator);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_Operator);

protected:
  APT_Operator();
  /*
     effect    Initializes this operator to the "undescribed" state.
               The describeOperator() function will be called by the
               framework when the step that this operator belongs to
               is checked.
  */

public:
  virtual ~APT_Operator();
  /*
     effect    Any attached datasets that are dynamically allocated
               and are not attached to another operator are deleted.
  */

  APT_ViewAdapter viewAdapter(int inputDS) const;
  APT_ViewAdapter inputAdapter(int inputDS) const;
  void setViewAdapter(const APT_ViewAdapter& map, int inputDS);
  void setInputAdapter(const APT_ViewAdapter& map, int inputDS);
  /*
     effect    Accesses the view adapter to be used to adapt the
               indicated dataset's actual schema to this operator's
               corresponding interface schema.
               The inputDS argument specifies which input dataset is
               being referred to.
               Defaults to default-constructed view adapter.
               The map object is copied.
               If the given input dataset already has a view
               adapter, it is replaced.
     note      When setting the view adapter, if any of its
               deprecated functions (relating to schema variables)
               have been called, then the transfer adapter is also
               set.
     requires  0 <= inputDS < the number of input dataset ports
               This requirement is checked at step check time.
  */

  APT_TransferAdapter transferAdapter(int inputDS) const;
  void setTransferAdapter(const APT_TransferAdapter& map, int inputDS);
  /*
     effect    Accesses the transfer adapter to be used to control
               transfers from schema variables in the interface
               schema.
               The inputDS argument specifies which input dataset is
```

```
              being referred to.
              Defaults to default-constructed transfer adapter.
              The map object is copied.
              If the given input dataset already has a transfer
              adapter, it is replaced.
   requires   0 <= inputDS < the number of input dataset ports
              This requirement is checked at step check time.
*/

APT_ModifyAdapter modifyAdapter(int outputDS) const;
APT_ModifyAdapter outputAdapter(int outputDS) const;
void setModifyAdapter(const APT_ModifyAdapter& map, int outputDS);
void setOutputAdapter(const APT_ModifyAdapter& map, int outputDS);
/*
   effect     Accesses the adapter used to adapt the indicated
              output dataset's latent schema to form the output
              dataset's actual schema.
              Defaults to default-constructed adapter.
              The map object is copied.
              If the given output dataset already has an adapter, it
              is replaced.
   requires   0 <= outputDS < the number of output dataset ports
              This requirement is checked at step check time.
*/

void attachInput(APT_DataSet* ds, int inputDS);
/*
   effect     Attaches the indicated dataset to the indicated input
              port of this operator.
   note       If ds is dynamically allocated, it becomes owned by
              this operator and will be deleted when this operator is
              destroyed (and the dataset is not attached to another
              operator).
   requires   ds non-null.
              0 <= inputDS < the number of input dataset ports.  This
              requirement is checked at step check time.
              If this dataset port's partitioning is eSame, then
              ds->isRDBMS() must be false.  This requirement is
              checked at step check time.
              ds must be an input-capable dataset.
              ds must not have been attached to any other operator's
              input.
              inputDS must not already have a dataset attached.
*/
const APT_DataSet* input(int inputDS) const;
APT_DataSet* input(int inputDS);
int inputPort(const APT_DataSet*) const;
/*
   effect     Returns the dataset attached to the indicated input
              port of this operator, or vice versa.
```

```
            Returns 0 (or -1) if no dataset is attached.
   requires  0 <= inputDS < the number of input dataset ports
            This requirements is checked at step check time.
*/

void attachOutput(APT_DataSet* ds, int outputDS);
/*
   effect    Attaches the indicated dataset to the indicated output
            port of this operator.
   note      If ds is dynamically allocated, it becomes owned by
            this operator and will be deleted when this operator is
            destroyed, (and the dataset is not attached to another
            operator).
   requires  ds non-null.
            0 <= outputDS < the number of output dataset ports
            The above requirement is checked at step check time.
            ds must be an output-capable dataset.
            ds must not have been attached to any other operator's
            output.
            outputDS must not already have a dataset attached.
*/
const APT_DataSet* output(int outputDS) const;
APT_DataSet* output(int outputDS);
int outputPort(const APT_DataSet*) const;
/*
   effect    Returns the dataset attached to the indicated output
            port of this operator, or vice versa.
            Returns 0 (or -1) if no dataset is attached.
   requires  0 <= outputDS < the number of output dataset ports
            This requirement is checked at step check time.
*/

void addNodeConstraint(const char* nodePool);
void addResourceConstraint(const char* resKind, const char* resPool="");
/*
   effect    Expresses a constraint governing the node(s) on which
            player(s) this operator may be placed.
            Initially, an operator's set of "available nodes"
            consists of all nodes in the default node pool.  The
            first added constraint replaces the initial set;
            subsequent constraints have the effect of reducing this
            operator's set of available nodes.
            Multiple constraints may be expressed, in which case
            the resulting set of available nodes is the
            intersection of the results of all of the constraints.
   note      An empty resPool argument selects for resources that
            are in the "default" pool for their resource kind (each
            resource kind has its own pool namespace).
            Constraints added to a composite operator are by
            default applied to the composite's sub-operators.
```

```
              Operators' describeOperator() can also express
              constraint(s), which are typically (but not
              necessarily) intersected with the user-specified
              constraints to form the operator's final set of
              available nodes.
   requires   Args non-null.
*/

APT_NodeSet availableNodes() const;
void setAvailableNodes(const APT_NodeSet&);
/*
   effect     Replaces the set of nodes on which this operator can
              run.  Calling this function overrides any prior calls
              to addNodeConstraint() and addResourceConstraint().
   note       If any operator's set of available nodes is empty,
              APT_Step::check() will fail unless setNodeMap() was
              called on it with a non-zero numberOfPartitions.
*/

void setNodeMap(const APT_Node* nodeVector, int numberOfPartitions);
/*
   effect     Specifies how many partitions an operator will have and
              exactly which node each partition will run on.  The
              first partition will run on nodeVector[0], the second
              partition on nodeVector[1], etc.  Multiple partitions
              can be assigned to the same node, e.g., if nodeVector[0]
              == nodeVector[1], then the first two partitions will run
              on the same node.  If this function is not called, or if
              numberOfPartitions is zero, the framework will choose
              the number of partitions and node assignments based on
              the set of available nodes.  If numberOfPartitions > 0,
              setNodeMap() overrides all calls (before and after) to
              addNodeConstraint(), addResourceConstraint() and
              setAvailableNodes().  When numberOfPartitions is zero,
              the only effect of the call is that setNodeMap() may
              never again be called with numberOfPartitions > 0.
               The node vector is copied and stored in the operator.
   note       In runLocally(), partitionNumber() returns a value
              between 0 and numberOfPartitions-1 (inclusive) if
              setNodeMap() was called.  Otherwise, the number is
              between 0 and the number of partitions (minus one)
              chosen by the framework.
              A node map added to a composite operator is *not*
              automatically applied to the composite's
              sub-operators.
   warning    Fixing the number of partitions of an operator can
              result in irreconcilable conflicts on its number of
              partitions if it has input or output datasets with
              eSame or preserve-partitioning.
   requires   numberOfPartitions >= 0.
```

```
            If numberOfPartitions > 0, then nodeVector != 0,
            and nodeVector has numberOfPartitions APT_Nodes.
            When this function is called multiple times, the
            numberOfPartitions and nodeVector must be the same.
            The nodes in the vector must be in the configuration
            file, but they need not be in the default node pool.
            This function may be called from describeOperator().
            This function may be called before APT_Step::check().
              If kind() is set to eSequential, setNodeMap() must not be
            called with numberOfPartitions > 1.
            If setNodeMap() was called with numberOfPartitions > 1, then
            kind() must not be set to eSequential.
*/

const APT_Node* nodeMap(int* nodeMapLength) const;
/*
   effect    Retrieves the node mapping in effect for this
             operator.  Returns 0 if there is no mapping.
   note      One way of establishing a node mapping is via
             setNodeMap().  If setNodeMap() is not called, then the
             Score Composer will compute a node mapping for this
             operator; this node mapping will be available in
             preRun(), runLocally(), and postRun().
*/

APT_Status initializeFromArgs(const APT_PropertyList& args);
/*
   effect    Initializes this operator's state from the indicated
             property list.
             Wraps the initializeFromArgs_() virtual (calling it in
             eInitial mode), and stores the argument property list
             in the rep (where it is serialized by the framework).
             See initializeFromArgs_() for more details.
*/

APT_PropertyList initializationArgs() const;
/*
   effect    Returns the property list that was passed to
             initializeFromArgs() (or setRuntimeArgs(), if that was
             called).
             Returns the empty list if neither of these functions
             was called.
*/


/* argument handling for new-form "wrapper-free" operators */

static APT_Operator* lookupAndInitializeFromArgv(int argc,
                                                 const char* const* argv,
                                                 APT_ErrorLog&,
```

```
                                                const char* ident=0);
static APT_Operator* lookupAndInitializeFromArgv(int argc,
                                                const APT_String* argv,
                                                APT_ErrorLog&,
                                                const char* ident=0);
/*
   effect    Interprets argv[0] as the OSH-name of this operator;
             uses the APT_OshNameRegistry (see osh_name.h) to obtain
             the C++ class of the operator, and also a description
             of the expected arguments, which are checked via an
             APT_ArgvProcessor object (see apt_util/argvcheck.h).
             If all goes well, the C++ class for the operator is
             instantiated, and initializeFromArgs() is called (with
             the argv-processor's output property list) before
             returning the operator pointer.
             If ident is non-null, then setIdent() is called on the
             instantiated operator object.
   errors    If the OSH name cannot be resolved, or the C++ class
             cannot be instantiated, then a null pointer is returned
             and information written to the passed error log object.
             Once a C++ class is instantiated for this operator, its
             error log object will be used instead of the passed
             error log object for subsequent error checks (argv
             processing, initializeFromArgs()).  Even if errors
             occur, a non-null operator pointer is returned (and
             its errorLog() will contain any errors that occurred).
   note      The argv should be allocated to argc elements; there is
             no need for an extra null argv element.
             The argc/argv information is stashed for use in
             accessArgv().
   requires  argc >= 1
             argv non-null
*/
static APT_Operator* lookupAndInitializeFromArgv(int argc,
                                                const char* const* argv,
                                                const char* ident=0);
static APT_Operator* lookupAndInitializeFromArgv(int argc,
                                                const APT_String* argv,
                                                const char* ident=0);
// variants of above, without user-supplied error log object

const APT_String* accessArgv(int* argc) const;
/*
   effect    Returns the argument list that was passed to
             lookupAndInitializeFromArgv().
             Returns 0 if this operator was not initialized via
             lookupAndInitializeFromArgv().
*/

// helper class for building up arg vectors
```

```
class ArgvAccum
{
public:
  ArgvAccum();
  ~ArgvAccum();

  void addArg(const char* arg);
  void reset();

  int argc() const { return argc_; }
  const APT_String* argv() const { return argv_; }

private:
  // prohibit copy/assign
  ArgvAccum(const ArgvAccum&);
  ArgvAccum& operator= (const ArgvAccum&);

  int argc_;
  int allocLen_;
  APT_String* argv_;
};


APT_String errorInformation() const;
/*
   effect    If reportError() has been called (or the errorLog()
             used), returns the error text.  Otherwise returns the
             empty string.
   DEPRECATED
*/

APT_String ident() const;
/*
   effect    Returns a string by which this operator may be
             identified.
             If setIdent() has not been called (or has been called
             with an empty string arg), returns the class name of
             this operator.
*/
void setIdent(const char*);
/*
   effect    Determines what ident() returns.
                  The value specified in this call is modified by the
             framework to disambiguate the value when the operator
             is bound into the step (via attachOperator or
             markSubOperator).  This disambiguation takes the form
             of a suffixed number (e.g. "(0)") if needed, and (for
             operators within composites) a suffixed "in <composite
             ident>".
             This is used by OSL to set an operator's ident() to be
```

```
               the operator's name as it appears on the command line.
     requires  Arg non null.
                    Must be called before the operator is bound into the
               framework (via attachOperator or markSubOperator).
  */

  enum Kind { eSequential, eParallel };

  void setRequestedKind(Kind);
  /*
     effect    Set the "requested kind" flag to the specified
               execution mode (sequential or parallel).
               If this function is not called, then the kind() defaults
               to eParallel unless describeOperator() calls setKind().
     note      This function is called when the OSH [seq/par] syntax is
               used.
               If this operator's describeOperator() calls setKind(),
               with a different value, then the setRequestedKind()
               call's value is disregarded and a warning issued.
               This function may be called multiple times, in which
               case the last (most recent) call dominates.
  */
  bool isRequestedKindSet() const;
  /*
     effect    Returns true if setRequestedKind() was called, otherwise
               false.
  */
  Kind requestedKind() const;
  /*
     effect    Returns the value passed to setRequestedKind().
     requires  isRequestedKindSet() must be true
  */

  Kind kind() const;
  /*
     effect    Gives the execution mode of this operator, as specified
               in setRequestedKind() or setKind().
               The default is eParallel.
     note      Before the step is checked, describeOperator() will not
               have been called, so kind() will reflect either the
               default setting (eParallel) or the value passed to
               setRequestedKind() if it was called.
  */

protected:
  enum InitializeContext
  {
    eInitial,           /* on conductor, prior to describeOperator()
                           and storing serialization */
    eRun                /* on player(s), after loading serialization
```

```
                        but prior to runLocally() */
};
virtual APT_Status initializeFromArgs_(const APT_PropertyList& args,
                                       InitializeContext context);
/*
   effect    If an operator supports OSL, it should override this
             function to interpret the supplied arguments.
             Any errors should be reported via reportError() and a
             APT_StatusFailed return value.
             The default implementation returns APT_StatusFailed and
             writes "Not an OSH-enabled operator" using
             reportError().
   note      This function is called twice: once on the conductor
             before describeOperator(), and then in the player(s)
             before runLocally().  The context argument can be used
             to distinguish between the two calls.
             Both calls receive the same property list (unless
             setRuntimeArgs() is called by this function or
             describeOperator()).
   returns   APT_StatusOk: this operator is happy with its arguments
             APT_StatusFailed: this operator found a problem with
               its arguments; more information may be supplied via
               reportError().
*/

virtual APT_Status describeOperator()=0;
/*
   effect    Must be overridden to do the following:
             - setKind() (optional)
             - setInputDataSets(), setOutputDataSets()
             - setInputInterfaceSchema(), setOutputInterfaceSchema()
             - call declareTransfer() as appropriate
             - setPartitionMethod() (optional)
             - setCollectionMethod() (optional)
             - setAvailableNodes(), addNodeConstraint(),
               addResourceConstraint() (optional)
             - setRuntimeArgs() (optional)
             - setCheckpointStateHandling() (optional)
             - setWorkingDirectory() (optional)
   note      This function is called once by the framework at
             step-check time.
             The operators in a step have their describeOperator()
             function called in dataflow execution order.  In other
             words, when a given operator A's describeOperator()
             function is called, any operators feeding operator A
             via virtual datasets will have been described already.
             The key consequence of this is that an operator's input
             datasets have valid schema when the operator's
             describeOperator() function is called.
   return    APT_StatusOk: All seems to be in order.
```

```
            APT_StatusFailed: Something is amiss.  See reportError().
*/

virtual APT_Status checkConfig();
/*
   effect    Called during step check, after all dataset schemas
             have been propagated through the operators and after
             dead field elimination has been performed, but before
             calling runLocally().
             This function can be overridden to provide additional
             schema checks, for example to require that multiple
             input datasets' interfaces are consistent.
             Default implementation returns APT_StatusOk.
   return    APT_StatusOk: All seems to be in order.
             APT_StatusFailed: Something is amiss.  See reportError().
*/

virtual APT_Status preRun();
/*
   effect    This function is called for all operators on the main
             program node during APT_Step::run().
             The default implementation is a nop that returns
             APT_StatusOk.
   return    APT_StatusOk: This operator has no problems.
             APT_StatusFailed: The step's run() is aborted, and is
               considered to have failed.  Information explaining
               the problem(s) should have been placed into this
               operator's errorLog().  No further preRun() calls are
               made.
   note      This function is called once by the framework at
             step-run time just prior to parallel execution
             startup.  In particular, any changes made to this
             operator's state will be available for transmission to
             the parallel nodes of this operator.
             This function is called for all operators, including
             composite operators, even though composite operators'
             runLocally() is not called during parallel execution.
             The operators in a step have their preRun() function
             called in a dataflow execution order.
             preRun() is only called once for a segmented step; it
             is not called for each step segment.
*/

virtual APT_Status runLocally()=0;
/*
   effect    Called by the framework as part of running the step
             that contains this operator.
             This function must be overridden to process records
             from the input dataset(s) (if any) and send records to
             the output dataset(s) (if any).
```

```
              Record data are typically read and written via
              accessors and aggregate cursor objects that are setup
              at the start of this function.
   note       This function is named runLocally() because it is
              called for each dataset partition for eParallel
              operators.
   return     APT_StatusOk: runLocally() ran to completion without any
                runtime errors.
              APT_StatusFailed: runLocally() is terminating (possibly early)
                with an error.  See reportError().
   throws     If this function throws any errors, the framework
              terminates the step immediately.
   requires   If this function returns APT_StatusOk, it must have
              consumed all records from all input datasets.
*/

virtual void postRun(APT_Status stepStatus);
/*
   effect     This function is called for all operators on the main
              program node after APT_Step::run().
              The default implementation is a nop.
   note       This function is called once by the framework upon
              completion of the step's run.
              The postRun() functions are called even if the step's
              execution was aborted due to a preRun() failure.
              This function is called for all operators, including
              composite operators, even though composite operators'
              runLocally() is not called during parallel execution.
              The operators in a step have their postRun() function
              called in a dataflow execution order.
              postRun() is only called once for a segmented step
              (upon step completion or abandonment); it is not called
              for each step segment.
*/

// notification functions
virtual void reportInputAttachment(const APT_DataSet* ds, int inputDS);
virtual void reportOutputAttachment(const APT_DataSet* ds, int outputDS);
/*
   effect     These functions are called by attachInput() and
              attachOutput() after the indicated action has been
              performed.
              Default implementation is a nop.
*/

void reportError(const char*);
/*
   effect     This function can optionally be called by
              initializeFromArgs_(), describeOperator(),
              checkConfig() or runLocally() before returning
```

```
              APT_StatusFailed.
              The string argument can contain any information that
              might be useful in describing why describeOperator(),
              checkConfig() or runLocally() is failing.
              This function may be called multiple times; the error
              strings are appended.
     requires  arg non-null.
     note      See also errorLog()
  */

  void reportWarning(const char* warn);
  /*
     effect    Similar to reportError(), but generates warning.
     requires  arg non-null.
     note      See also errorLog()
  */

public:
  APT_ErrorLog& errorLog();
  /*
     effect    Returns access to this operator's internal error log
               object.
     note      Any traffic generated on this log will be identified as
               originating from this operator.
  */

  static APT_Operator* currentOperator();
  /*
     effect    Returns the operator instance that is currently
               executing, or 0 if there is no operator currently in
               the dynamic scope.
     note      This is non-null during runLocally().
               The combined operator controller properly updates this
               pointer as control passes from combinee to combinee.
  */


  /* TBD: need a way for ACI client to pass information to operators,
     individually for each player? */


  /**** functions called from describeOperator() overrides ****/

  /* These functions should only be called from within
     describeOperator().  They should never be called from within
     runLocally() or checkConfig(). */

protected:
  void setKind(Kind);
  /*
```

```
    effect    Specifies whether this operator is sequential or
              parallel.
              The default is eParallel or whatever was passed to
              setRequestedKind().
              If setRequestedKind() was called with a value different
              than that passed to setKind(), a warning is issued and
              the setKind() value overrides the setRequestedKind()
              value.
    requires  This function may be called at most once.
  */

  void setInputDataSets(int);
  void setOutputDataSets(int);
  /*
    effect    Specifies the number of input and output dataset
              attachments expected by this operator.
              There is no default.  Each of these functions must be
              called exactly once.
    requires  arg >= 0
  */

  bool checkWasSuccessful() const;
  /*
    effect    Tells if this operator has been successfully checked.
  */

public:
  int inputDataSets() const;
  int outputDataSets() const;
  // requires: set{Input,Output}DataSets() must have been called

protected:
  APT_Schema inputInterfaceSchema(int inputDS) const;
  void setInputInterfaceSchema(const APT_Schema& schema, int inputDS);
  APT_Schema outputInterfaceSchema(int outputDS) const;
  void setOutputInterfaceSchema(const APT_Schema& schema, int outputDS);
  /*
    effect    Accesses the interface schema to be used for the
              indicated dataset.
              The schema object is copied.
              If the given dataset already has an interface schema,
              it is replaced.
              An interface schema must be set for each of an
              operator's inputs and outputs.
    requires  0 <= inputDS < the number of input dataset ports
              0 <= outputDS < the number of output dataset ports
              schema.isInterfaceSchema() is true.
              When getting an interface schema for a given dataset port,
              it must have been set first.
  */
```

```
  int declareTransfer(const APT_FieldSelector& inputComponent,
                      const APT_FieldSelector& outputComponent,
                      int inputDS, int outputDS);
  /*
     effect    This function tells the framework that this operator's
               runLocally() can call transfer() under certain
               circumstances.
               The component arguments are evaluated relative to the
               interface schemas that wers specified in the call to
               set{Input,Output}InterfaceSchema().
     returns   An index value that can be passed to transfer().  The
               first call to declareTransfer() returns 0; subsequent
               calls return successive integer values.
     note      All schema variables in each of this operator's input
               and output interface schemas should be part of at least
               one declared transfer (a schema variable cannot be
               accessed or manipulated by an operator in any manner
               other than being transferred).
     requires  0 <= inputDS < the number of input dataset ports
               0 <= outputDS < the number of output dataset ports
               inputComponent (outputComponent) must refer to a field
               of the indicated input's (output's) interface schema of
               type "var".
  */

public:
  enum PartitionMethod
  {
    eAny,
    eRoundRobin,
    eRandom,
    eSame,
    eEntire,
    eOther
  };
  // See operator<<() below for printing

protected:
  void setPartitionMethod(PartitionMethod m, int inputDS);
  /*
     effect    Specifies the partitioning method to be used on the
               indicated input dataset.
               By default, eAny is the partition method.
     requires  0 <= inputDS < the number of input dataset ports
               kind() must not be set to eSequential.
               m must not be eOther
  */
  void setPartitionMethod(APT_Partitioner* part,
                          const APT_ViewAdapter& adapter, int inputDS);
```

```
  /*
     effect    Specifies eOther partitioning for the indicated input
               dataset, using the supplied partitioner object.
               The adapter argument is used to adapt this operator's
               indicated input's interface schema to the partitioner's
               input interface schema.
               The adapter object is copied.
     requires  0 <= inputDS < the number of input dataset ports
               kind() must not be set to eSequential.
               part non-null
               part should be dynamically allocated.  It will be
               deleted by the framework.
  */

public:
  static APT_Status lookupPartitionMethod(const APT_PropertyList& wrapperOut,
                                          APT_Partitioner* *partObj,
                                          PartitionMethod* partMethod,
                                          APT_ErrorLog&);
  /*
     effect    Figures out the partitioning method described in the
               wrapperOut.
               The wrapperOut arg is expected to be the output of a
               partitioner's OSH wrapper.  The partition method (or
               partitioner object) is chosen (and initialized if
               necessary) according to the wrapperOut arg.
               The partObj argument, if returned as non-null, is
               dynamically allocated and must either be deleted or
               passed to setPartitionMethod().
     returns   APT_StatusOk: wrapperOut is valid; either partObj or
                 partMethod has been updated.
               APT_StatusFailed: wrapperOut is invalid; additional
                 information is written to the error log object.
  */

  enum CollectionMethod
  {
    eCollectAny,                /* operator consumes its input on a
                                   first-arrived basis. */
    eCollectRoundRobin,         /* records from incoming partitions
                                   are consumed in a round-robin manner;
                                   as particular partitions reach EOF, the
                                   remaining partitions are
                                   round-robined. */
    eCollectOrdered,            /* all records from the zeroth
                                   partition are consumed, followed by
                                   all records from the next
                                   partition, etc, until all input
                                   records have been consumed. */
    eCollectOther               /* an object derived from the APT_Collector
```

```
                                    base class is used to determine the order
                                    in which records are processed */
  };
  // See operator<<() below for printing

protected:
  void setCollectionMethod(CollectionMethod m, int inputDS);
  /*
     effect    Specifies the collection method to be used on the
               indicated input dataset.
               By default, eCollectAny is the collection method.
     requires  0 <= inputDS < the number of input dataset ports
               kind() must have been set to eSequential.
               m must not be eCollectOther
  */
  void setCollectionMethod(APT_Collector* coll,
                           const APT_ViewAdapter& adapter, int inputDS);
  /*
     effect    Specifies eOther collection method for the indicated input
               dataset, using the supplied collector object.
               The adapter argument is used to adapt this operator's
               indicated input's interface schema to the collector's
               input interface schema.
               The adapter object is copied.
     requires  0 <= inputDS < the number of input dataset ports
               kind() must have been set to eSequential.
               coll non-null
               coll should be dynamically allocated.  It will be
               deleted by the framework.
  */

public:
  static APT_Status lookupCollectionMethod(const APT_PropertyList& wrapperOut,
                                           APT_Collector* *collObj,
                                           CollectionMethod* collMethod,
                                           APT_ErrorLog&);
  /*
     effect    Figures out the collection method described in the
               wrapperOut.
               The wrapperOut arg is expected to be the output of a
               collector's OSH wrapper.  The collection method (or
               collector object) is chosen (and initialized if
               necessary) according to the wrapperOut arg.
               The collObj argument, if returned as non-null, is
               dynamically allocated and must either be deleted or
               passed to setCollectionMethod().
     returns   APT_StatusOk: wrapperOut is valid; either collObj or
                 collMethod has been updated.
               APT_StatusFailed: wrapperOut is invalid; additional
                 information is written to the error log object.
```

```
  */

protected:
  APT_Schema inputConcreteSchema(int inputDS) const;
  /*
     effect    Returns the concrete schema associated with the
               indicated attached dataset.
     note      You probably want to use viewAdaptedSchema() instead.
     requires  0 <= inputDS < the number of input dataset ports
  */

  APT_Schema viewAdaptedSchema(int inputDS) const;
  APT_Schema inputAdaptedSchema(int inputDS) const;
  /*
     effect    Returns the schema computed by taking the concrete
               schema associated with the indciated attached dataset
               and "projecting" it through the view adapter, if any.
               See APT_ViewAdapter::viewAdaptedSchema() for
               details.
     requires  0 <= inputDS < the number of input dataset ports
  */

  void setPreservePartitioningFlag(int outputDS);
  void clearPreservePartitioningFlag(int outputDS);
  /*
     effect    Sets or clears the preserve partitioning (PP) flag for
               the indicated output dataset.
     note      This action overrides the PP flag that would be
               propagated by the framework.  It does not override a PP
               flag set or cleared by the user via APT_DataSet
               functions.
     requires  0 <= outputDS < the number of output dataset ports
  */

  void setRuntimeArgs(const APT_PropertyList& args);
  /*
     effect    Can be called from describeOperator() to specify an
               initialization property list to be passed to
               initializeFromArgs_() at eRun time.
     note      This is intended to be useful for the case of an
               OSL-enabled operator used via its C++ interface, where
               the operator's implementation does not use persistence
               but rather uses initializeFromArgs_() at runtime.
  */

public:
  enum CheckpointStateHandling
  {
    eDoesNotHandleState,        /* operator does not contain logic
                                   needed to manage state across step
```

```
                                   segment boundaries; steps
                                   containing such operators are not
                                   restartable.  This is the default. */
    eNoState,                   /* operator does not need to manage
                                   any state across step segment
                                   boundaries. */
    eSimpleState,               /* operator manages state across step
                                   segment boundaries using
                                   inputCheckpointData() and
                                   outputCheckpointData() interface */
    eStateWithCleanup           /* as eSimpleState; in addition,
                                   operator requires a cleanup segment
                                   if a step is abandoned */
  };
protected:

  void setCheckpointStateHandling(CheckpointStateHandling h);
  CheckpointStateHandling checkpointStateHandling() const;
  /*
     effect    Can be called by describeOperator() to specify how this
               operator manages step segment boundaries.
               The default is eDoesNotHandleState, indicating that
               this operator cannot be used in a restartable step.
  */


  /**** functions called from runLocally() overrides ****/

  /* These functions should only be called from within runLocally().  They
     should never be called from within describeOperator() or
     checkConfig(), and should certainly never be called by the user
     of an operator.
     An exception is that checkConfig() may setup accessors and
     aggregate cursor (for the purpose of checking vector lengths
     and aggregate tags).
  */

  int partitionNumber() const;
  /*
     returns   Partition number of this player.
               For a parallel operator with k partitions,
                 0 <= return value < k
               Returns 0 for a sequential operator.
     requires  Called only from within the dynamic scope of runLocally()
     advanced  This partition id is guaranteed to correspond to the
               vector indices used in APT_Operator::setNodeMap().
  */

  int partitionCount() const;
  /*
```

```
    returns   The number of partitions in the operator.
              Returns 1 for a sequential operator.
    requires  Called only from within the dynamic scope of runLocally()
  */

  int inputPartitionCount(int inputDS) const;
  /*
    effect    Returns the degree of parallelism of the operator
              feeding the indicated input to this operator.
    note      If the indicated input is a persistent data set (with
              no feeding repartition operator), returns the same
              answer as partitionCount().
    requires  0 <= inputDS < the number of input dataset ports
              Called only from within the dynamic scope of runLocally()
  */

  int inputPartition(int inputDS) const;
  /*
    effect    Returns the number of the partition that the current
              record of inputDS came from, or -1 if the input partition
              can't be determined.
    note      The input partition can't be determined for persistent
              datasets, before a record has been read, or after EOF.
    requires  0 <= inputDS < the number of input dataset ports
              Called only from within the dynamic scope of runLocally()
  */

  int outputPartitionCount(int inputDS) const;
  /*
    effect    Returns the degree of parallelism of the operator
              consuming the indicated output of this operator.
    note      If the indicated output is a persistent data set (with
              no writing repartition operator), returns the same
              answer as partitionCount().
    requires  0 <= outputDS < the number of output dataset ports
              Called only from within the dynamic scope of runLocally()
  */


protected:
  void setWorkingDirectory(bool alter,
                           const char * directory = 0);
  /*
    effect    Controls whether or not the working directory will be
              set to any particular value before runLocally is
              called.

              If called with alter=false, the working directory will
              be indeterminate when runLocally is called.
```

```
                If called with alter=true, directory=0, the working
                directory for runLocally will be set to the current
                working directory of the Orchestrate main program. If
                this proves impossible, the step will fail.

                If called with alter=true, directory="string", the
                working directory for runLocally will be set to the
                absolute path given by the string. If this is not
                possible, the step will fail.

                If this is not called before describeOperator returns,
                the default is as if setWorkingDirectory(true, 0) were
                called.

      requires  If alter=false, directory must be zero.
                If alter=true and directory != 0, the string pointed
                to by directory must be an absolute path name (one that
                begins with '/').
   */

   bool workingDirectorySet() const;
   /*
      returns   The value passed to setWorkingDirectory as "alter".
   */

   APT_String workingDirectory() const;
   /*
      returns   The value passed to setWorkingDirectory as "directory.
                If directory was 0, an empty string is returned.
   */

public:
   /* These functions are made public for the benefit of operators
      whose runLocally() is structured such that these functions will
      be called indirectly (by a function called from runLocally() and
      handed a pointer to the running operator instance).

      This could be done by friend decls in the derived operator class,
      but if polymorphism is desired, and extra level of derivation
      would be needed and this seems a bit artificial.  So we just make
      these functions public; these functions never make sense to be
      called by an operator's user so this probably won't lead to
      confusion.

      TBD: add run-time enforcement that these functions must be called
      from within the operator's runLocally() activation.
   */

   void setupInputCursor(APT_InputCursor* cur, int inputDS);
   void setupOutputCursor(APT_OutputCursor* cur, int outputDS);
```

```
/*
   effect    Initializes the given cursor object so it can be used
             to read (write) field data from (to) the indicated
             dataset.
   note      The cursor's schema() is the interface schema of the
             indicated dataset.
   requires  cur non-null
             cur->isSetup() should be false upon entry.
             The indicated dataset port must not already have a
             cursor.
             0 <= inputDS < the number of input dataset ports
             0 <= outputDS < the number of output dataset ports
*/

void transfer(int transferIndex);
/*
   effect    Causes the framework to perform the indicated transfer
             operation.
   requires  transferIndex must be a value returned by one of the
             calls to declareTransfer().
             APT_InputCursor::getRecord() must have been called and
             returned true for the dataset that sources the transfer.
*/

void deferredTransfer(int transferIndex);
/*
   effect    Causes the framework to perform the indicated transfer
             operation.
   note      Unlike transfer(), this routine does not necessarily
             copy information to the output record; in many cases the
             output record is made to reference material in the
             input record, deferring the actual copy until
             putRecord().  This can result in improved performance.
   requires  Same as transfer().
             The input record must not be disturbed until after
             putRecord() is called.
*/

void transferAndPutRecord(int transferIndex);
/*
   effect    Equivalent to deferredTransfer(transferIndex) followed
             by a putRecord() on the output cursor that is the
             destination of the transfer.
   requires  Same as transfer().
             An output cursor must have been set up, even if it is
             not used directly.
*/

bool isTransferBufferFixedLength(int transferIndex) const;
/*
```

```
   effect    Returns true *only if* the given transfer's
             getTransferBufferSize() always returns the same
             value.
   note      This function may be called from the commandLine()
             function of a subprocess operator.
   requires  transferIndex must be a value returned by one of the
             calls to declareTransfer().
*/
APT_UInt32 getTransferBufferSize(int transferIndex) const;
/*
   effect    Returns the buffer size required for a call to
             transferToBuffer(transferIndex).
   requires  transferIndex must be a value returned by one of the
             calls to declareTransfer().
             APT_InputCursor::getRecord() must have been called and
             returned true for input's dataset; this requirement is
             waived if isTransferBufferFixedLength(transferIndex) is
             true.
*/
void transferToBuffer(char* buf, int transferIndex) const;
/*
   effect    Causes the framework to copy the field data associated
             with an input schema variable to the supplied buffer.
             The input schema variable that sources the indicated
             transfer is the schema variable whose fields are copied
             to the buffer.
   note      This transfer operation can be completed by
             transferFromBuffer().
   requires  transferIndex must be a value returned by one of the
             calls to declareTransfer().
             APT_InputCursor::getRecord() must have been called and
             returned true for input's dataset.
             buf must be non-null and allocated to hold at least
             getTransferBufferSize(transferIndex) bytes.
*/

void transferFromBuffer(const char* buf, int transferIndex);
/*
   effect    Causes the framework to complete the indicated transfer
             started by a preceding transferToBuffer().
   requires  buf non-null
             buf must refer to a buffer that was filled using
             transferToBuffer() using the same transferIndex as in
             this call.  The alignment of the buffer is not
             important.
             The contents of this buffer must not have been modified
             by the client.
             The contents of this buffer must be stable until
             putRecord() is called.
             transferIndex must be a value returned by one of the
```

```
              calls to declareTransfer().
              The in-record alignment of the bound input and output
              interface schema variables must be identical to the
              schema variables used when the data was
              transferToBuffer()-ed.
  */

  APT_Int32 transferBufferOffset(const APT_FieldSelector& component,
                                 int transferIndex) const;
  /*
     effects   If the the specified input interface field meets
               these requirements:
               - present in buffers written via transferToBuffer
                 for the specified transfer, and
               - the input field is fixed-length, and
               - the inputfield is not a var-length vector, and
               - the input field is present at a fixed offset in the
                 transfer buffer, and
               - the input field is not being converted in any way
                 between its concrete form and its interface form
               then its offset in the transfer buffer is returned;
               otherwise -1 is returned.
     note      This function may be called from the commandLine()
               function of a subprocess operator.
     requires  The given component must be part of this operator's
               input interface schema (for the given transfer).
               The given component must be top-level (not in a subrec
                 or tagged).
               transferIndex must be a value returned by one of the
               calls to declareTransfer().
  */

protected:
  bool isSegmented() const;
  /*
     effect    Tells if this operator's step is segmented, which will be
               true iff setupCheckpointing() was called for it.
     requires  This function may only be called from the dynamic scope
               of describeOperator() and runLocally().
  */

  int segmentNumber() const;
  /*
     effect    Tells which segment is being executed.
               If isSegmented() is false, always returns 0.
     requires  This function may only be called from the dynamic scope
               of runLocally().
  */

  bool isCleanupSegment() const;
```

```
/*
   effect    Tells if this segment is being executed for the
             purposes of abandon() giving stateful operators a
             chance to clean up auxiliary state.
   note      In a cleanup segment, all operators' input data sets
             appear to be empty (getRecord() returns false), and
             calling putRecord() on any output data set is a "loud
             nop".
   requires  This function may only be called from the dynamic scope
             of runLocally().
*/

APT_Archive& outputCheckpointData();
/*
   effect    Can be used by runLocally() to write this paritition's
             checkpoint information describing the progress made by
             this segment of execution.  The returned archive will
             be in eStoring mode.
             This archive can be used by runLocally() to store
             checkpoint data before returning APT_StatusOk.
             Checkpoint data stored before an APT_StatusFailed
             return (in this or any other player) is not retained.
   note      The amount of information placed into a checkpoint
             archive should be modest (should fit comfortably into
             memory).  If large amounts of checkpoint data are
             required, then scratch files should be used, and the
             archived checkpoint data should contain the scratch
             file names (these are called auxiliary checkpoint data
             files).
   requires  checkpointStateHandling() must be eSimpleState or
             eStateWithCleanup.
*/

APT_Archive& inputCheckpointData();
/*
   effect    Can be used by runLocally() to access this partition's
             checkpoint data written out by the previous segment's
             execution.
             This information is typically needed so that the
             current segment knows where to "pick up" processing
             that was suspended at the end of the previous segment.
             The returned archive will be in eLoading mode.
   note      For segment 0, the returned archive is empty.  The
             returned archive is also empty if the step is not
             segmented.
   requires  checkpointStateHandling() must be eSimpleState or
             eStateWithCleanup.
*/

void runAnotherSegment();
```

```
/*
   effect    May be called by an operator prior to returning from
             runLocally(), to indicate that another segment should
             be run for the step.
   note      If no operator calls this function, then the
             segmentation of the input data sets governs how many
             segments are run for this step.
             Multiple calls are equivalent to one call.
   requires  May only be called from the dynamic scope of operators'
             runLocally().
             APT_Step::isSegmented() must be true.
   TBD: in segments following an operator becoming done, it is a
   "loud nop" to call putRecord() or runAnotherSegment()?
*/

void setInputPartitionEOFIndicator(int inputDS, bool *indicator);
/*
   effect    Sets the given indicator as the place to report EOF events
             on input partitions. Query inputPartitionEOF to find out
             which partitions are at EOF.
   requires  May only be called from the dynamic scope of operators'
             runLocally().
*/

bool inputPartitionEOF(int inputDS, int inputPartition) const;
/*
   effect    Tells whether the given input partition of the given
             dataset is at EOF.
   requires  May only be called from the dynamic scope of operators'
             runLocally().
*/

bool areInputDataSetsFinished() const;
/*
   effect    Tells if all of this operator's inputs have reached
             EOF, and will not have input records even if more step
             segments are executed.
   requires  May only be called from the dynamic scope of operators'
             runLocally().
*/


/* describeOperator hook functions

   Note: These are intended for Torrent internal use
         in specialized operator derivations (such as
             APT_SubProcessOperator). For this reason they are
         NOT documented, and their use is not supported.

   These provide a mechanism for base classes to get control
```

```
   just before and/or just after describeOperator() is called.
   For this to work, each class that overrides either function
   must call that function in its own base class, like this:

   APT_Status NewBase::preDescribeOperator()
   {
      APT_Status s = OldBase::preDescribeOperator();
      if (s != APT_StatusOk) return s;
      // do my processing
      if (someErrorOccurs)
         {
                Important: generate useful message
            reportError("NewBase: something bad happened.");
            return APT_StatusFailed;
         }
      return APT_StatusOk;
   }

   APT_Status NewBase::postDescribeOperator()
   {
      // do my processing
      if (someErrorOccurs)
         {
            // Important: generate useful message
            reportError("NewBase: something bad happened.");
            return APT_StatusFailed;
         }
      return OldBase::postDescribeOperator();
   }

   The default implementations do nothing more than
   return APT_StatusOk.
*/

virtual APT_Status preDescribeOperator();
virtual APT_Status postDescribeOperator();
/*
  effect      See above.
  returns     APT_StatusOk if processing should continue;
              otherwise, APT_StatusFailed.
*/

void dump(ostream& os) const;
APT_String dumpStr() const;
/*
   effect     Writes a textual representation of the op on the
              specified output stream or returns it as a string.
*/
```

```
private:
  friend class APT_SC; // access to scInterface()
  friend class APT_OperatorRep;
  friend class APT_OperatorSC;
  friend class APT_StepRep;
  friend class APT_CompositeOperator;
  friend class APT_OSL;         // for printing interface schemas
  friend class APT_Operator_UT; // for testing

  // prohibit copying
  APT_Operator(const APT_Operator&);
  APT_Operator& operator= (const APT_Operator&);

  // for SC
  const APT_OperatorSC* scInterface() const;
  APT_OperatorSC* scInterface();
  /*
     effect    Returns a pointer to an object providing supplemental
               interface to this operator.  For the step compiler
               implementation.
     note      The caller must not delete the returned pointer.
  */

  void stashArgv(int argc, const APT_String* argv); // copies argv


public:                         // bogus...
  APT_OperatorRep* rep_;
};

ostream& operator<<(ostream& os, APT_Operator::PartitionMethod cm);
ostream& operator<<(ostream& os, APT_Operator::CollectionMethod cm);
// convenience for printing the methods.

#endif // APT_OPERATOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1998 Torrent Systems, Inc.  All rights reserved.


#define APT_FRAMEWORK_VERSION "1"

#ifndef APT_OSH_NAME_H
#define APT_OSH_NAME_H


#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif
#include <apt_util/errlog.h>

/* Registry for associating a C++ class (operator, partitioner, or
   collector) with its corresponding OSH command-line name and usage
   string.
*/

class APT_OshNameRegistryRep;

class APT_OshNameRegistry
{
public:
  static APT_OshNameRegistry& get();
  // singleton global instance

  void registerName(const char* className, const char* oshName,
                    const char* argsDescription);
  /*
     effect    Registers the given className/oshName association.
     requires  Args must be non-null.
               className and oshName must both be unique among all
               registered entries.
  */

  struct Entry
  {
    APT_String className_;
    APT_String oshName_;
    APT_String argsDescription_;
  };
  // See APT_ArgvProcessor (in apt_util/argvcheck.h)
  // for syntax of argsDescription_
```

```
  Entry lookupByOshName(const char* oshName, APT_ErrorLog &log) const;
  /*
     effect    Returns an entry corresponding to the given OSH name.
               If there is no registered item with the given OSH name
               (case-sensitive), the returned Entry has all empty
               strings.
     requires  Arg must be non-null.
  */

  int numEntries() const;
  Entry* entries() const;
  /*
     effect    Enumerates all currently-registered entries, in
               unspecified order.
               The user is responsible for delete[]ing the returned
               vector.
  */

  ~APT_OshNameRegistry();

private:
  APT_OshNameRegistry();        // use get()

  APT_OshNameRegistry(const APT_OshNameRegistry&);
  APT_OshNameRegistry& operator= (const APT_OshNameRegistry&);

  APT_OshNameRegistryRep* rep_;
};

/*
  macro for conveniently generating the string used to create the
  file used to populate APT_OperatorRegistry. Includes the framework
  version number for versioning.

  Called automatically by APT_DEFINE_OSH_NAME for argv-checked operators.

  Call explicitly once per operator at file scope for wrapped operators.

   C: class name (token); O: osh name (token);

   Note that for wrapped operators, the osh name is the wrapper name
   without the ".op" or ".wrp" extension.

   APT_REGISTER_OPERATOR(APT_CheckConfigOperator, check_config);

   This is char[] because some compilers won't initialize a char*
   with a string, and others elide unused const char * on non-debug
```

```
   builds. char[] is what's used for Ident.C, and it works, so it's
   used here.

*/
#define APT_REGISTER_OPERATOR(C, O) \
char registerOsh_ ## C ## _ ## O ## _ident[] = \
    "$OshOperatorRegister: " #O " " APT_FRAMEWORK_VERSION " $";

/* macro for conveniently populating the APT_OshNameRegistry.  Use
   at file scope.

   C: class name (token); O: osh name (token); U: arglistDesc (quoted
   string).  Example:

     APT_DEFINE_OSH_NAME(APT_TsortOperator, tsort, "{key=...}");
*/

#define APT_DEFINE_OSH_NAME(C, O, U)                    \
  APT_REGISTER_OPERATOR(C, O);                          \
  static int registerOsh_ ## C ## _ ## O ## _func()     \
  { APT_OshNameRegistry::get().registerName(#C, #O, U); \
    return 0; }                                         \
  static int registerOsh_ ## C ## _ ## O ## _dummy =    \
    registerOsh_ ## C ## _ ## O ## _func()

#endif // APT_OSH_NAME_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_PARTITIONER_H
#define APT_PARTITIONER_H

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_ISDYNAMIC_H
#include <apt_util/isdynamic.h>
#endif


class APT_Schema;
class APT_InputAccessorBase;
class APT_InputInterface;
class APT_FieldSelector;
class APT_PartitionerRep;
class APT_PartitionerSC;
class APT_Operator;
class APT_OperatorRep;
class APT_DataSetRep;
class APT_StepRep;
class APT_PropertyList;
class APT_ErrorLog;

class APT_Partitioner : public APT_Persistent, public APT_IsDynamic
{
  APT_DECLARE_RTTI(APT_Partitioner);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_Partitioner);

protected:
  APT_Partitioner();
  /*
     effect    Initializes this partitioner to the "undescribed" state.
               The describePartitioner() function will be called by the
```

```
                framework when first needed.
  */

public:
  virtual ~APT_Partitioner();

  APT_Status initializeFromArgs(const APT_PropertyList& args);
  /*
     effect    Initializes this partitioner's state from the indicated
               property list.
               Wraps the initializeFromArgs_() virtual (calling it in
               eInitial mode), and stores the argument property list
               in the rep (where it is serialized by the framework).
               See initializeFromArgs_() for more details.
     note      Although this is designed to be an OSL entry point,
               it can also be used from C++ ADI programs to initialize
               OSL-aware partitioners via their command-line argument
               interface.  The args property list should be of the
               form produced by this partitioner's OSL wrapper.
  */

  APT_PropertyList initializationArgs() const;
  /*
     effect    Returns the property list that was passed to
               initializeFromArgs() (or setRuntimeArgs(), if that was
               called).
               Returns the empty list if neither of these functions
               was called.
  */


  APT_String errorInformation() const;
  /*
     effect    If reportError() has been called (or the errorLog()
               used), returns the error text.  Otherwise returns the
               empty string.
     DEPRECATED
  */

  APT_String ident() const;
  /*
     effect    Returns a string by which this partitioner may be
               identified.
               If setIdent() has not been called (or has been called
               with an empty string arg), returns the class name of
               this partitioner.
               ident() is used to distinguish between multiple instances
```

```
                of the same partitioner in a step.
  */
  void setIdent(const char*);
  /*
     effect    Determines what ident() returns.
     requires  Arg non null.
  */

protected:
  enum InitializeContext
  {
    eInitial,           /* on conductor, prior to describePartitioner()
                           and storing serialization */
    eRun                /* on player(s), after loading serialization
                           but prior to setupInputs() */
  };
  virtual APT_Status initializeFromArgs_(const APT_PropertyList& args,
                                         InitializeContext context);
  /*
     effect    If an partitioner supports OSL, it should override this
               function to interpret the supplied arguments.
               Any errors should be reported via reportError() and a
               APT_StatusFailed return value.
               The default implementation returns APT_StatusFailed and
               writes "Not an OSH-enabled partitioner" using
               reportError().
     note      This function is called twice: once on the conductor
               before describePartitioner(), and then in the player(s)
               before runLocally().  The context argument can be used
               to distinguish between the two calls.
               Both calls receive the same property list (unless
               setRuntimeArgs() is called by this function or
               describePartitioner()).
     returns   APT_StatusOk: this partitioner is happy with its arguments
               APT_StatusFailed: this partitioner found a problem with
                 its arguments; more information may be supplied via
                 reportError().
  */

  virtual APT_Status describePartitioner()=0;
  /*
     effect    Must be overridden to describe this partitioner's
               interface, by calling setInputInterfaceSchema().
               Note that a partitioner can have no interface schema,
               in which case describePartitioner() can be overridden
               to do nothing.
     note      This function is called once by the framework at
```

```
            step-check time.
            The this partitioner's operator's describeOperator()
            function will have been called already.
   return   APT_StatusOk: All seems to be in order.
            APT_StatusFailed: Something is amiss.  See reportError().
*/

virtual APT_Status setOutputPartitions(int numPartitions);
/*
   effect   Can be overridden by a partitioner to find out how
            many partitions the consuming operator has.
            This function is called once by the framework at
            step-check time, after describePartitioner().
            The base implementation just returns APT_StatusOk.
   return   APT_StatusOk: All seems to be in order.
            APT_StatusFailed: Something is amiss.  See reportError().
*/

virtual APT_Status setupInputs(int numPartitions)=0;
/*
   effect   Must be overridden to attach this partitioner's
            accessor(s), by calling setupInputAccessor().
            The numPartitions argument is guaranteed to be
            positive.  It is the same as what will be passed to
            partitionInput().
   note     This function will be called by the framework prior to
            the first call to partitionInput().
            Note that this function is called from the first
            putRecord() call (in each partition of each writing
            operator).
            Unlike operators, partitioners typically contain their
            accessor objects as private data members, initialized
            in setupInputs() and used in partitionInput().
            If this is a keyless partitioner, this function should
            be overridden to do nothing.
   return   APT_StatusOk: All seems to be in order.
            APT_StatusFailed: Something is amiss.  See reportError().
*/

void duplicateInput();
/*
   effect   If called by partitionInput(), causes the current
            record to be re-submitted to partitionInput() after the
            current call to partitionInput() returns.
   note     This applies only to the current partitionInput() call.
   requires This routine may only be called during a partitionInput()
            call.
```

```
*/

void discardInput();
/*
   effect    If called by partitionInput(), causes the current
             record to be discarded.
   note      This applies only to the current partitionInput() call.
   requires  This routine may only be called during a partitionInput()
             call.
*/

virtual int partitionInput(int numPartitions)=0;
/*
   effect    Called by the framework to assign a partition number to
             a record.
             This function must be overridden to examine the current
             input record (via the accessor objects initialized in
             setupInputs()) and return a partition number for it.
             The numPartitions argument is guaranteed to be
             positive.  It specifies the number of partitions
             available for the record being examined.
   requires  The returned value must satisfy:
               0 <= retval < numPartitions
             Alternately, if retval==-1, a fatal error is signalled
             by the framework (in which case it is advised that the
             partitioner will have placed relevant information into
             its errorLog()).
*/

virtual APT_Status startingConsumerPartition(int partNum);
/*
   effect    Called by the framework in each partition of the
             consuming operator that has this partitioner for one of
             its input datasets.
             The argument indicates which consuming partition is
             about to be runLocally()ed.
             The default implementation is a nop.
   return    APT_StatusOk: All seems to be in order.
             APT_StatusFailed: Something is amiss.  See reportError().
*/

APT_Schema inputInterfaceSchema() const;
/*
   effect    Returns this partitioner's input interface schema.
*/

APT_Schema inputConcreteSchema() const;
```

```
/*
   effect    Returns the concrete schema associated with the
             input dataset being partitioned by this partitioner.
   note      If this partitioner is in an operator, this function
             returns the input interface schema of the operator.
             If this partitioner is attached directly to a data set,
             this function returns the concrete schema of the data
             set.
*/

APT_Schema viewAdaptedSchema() const;
/*
   effect    Returns the schema computed by taking the concrete
             schema and "projecting" it through the view adapter
             associated with this partitioner, if any.
             See APT_ViewAdapter::viewAdaptedSchema() for
             details.
*/

/**** functions called from describePartitioner() overrides ****/

/* These functions should only be called from within
   describePartitioner().  It should never be called from within
   setupInputs() or partitionInput(). */

void setInputInterfaceSchema(const APT_Schema& schema);
/*
   effect    Sets the partitioner-centric schema to be used for this
             partitioner's input records.
             The schema object is copied.
             This is optional: it is legal for a partitioner to have
             no interface schema.  Such partitioners are called
             "keyless" partitioners.
   requires  The schema must consist of zero or more non-aggregate,
             non-vector fields.  No schema variables.
             This function may only be called from within
             describePartitioner(), and may only be called once.
*/

void setRuntimeArgs(const APT_PropertyList& args);
/*
   effect    Can be called from describePartitioner() to specify an
             initialization property list to be passed to
             initializeFromArgs_() at eRun time.
   note      This is intended to be useful for the case of an
             OSL-enabled partitioner used via its C++ interface, where
             the partitioner's implementation does not use persistence
```

```
                but rather uses initializeFromArgs_() at runtime.
  */

  void reportError(const char*);
  /*
     effect    This function can optionally be called by
               describePartitioner() or setupInputs() before returning
               APT_StatusFailed.
               The string argument can contain any information that
               might be useful in describing why describePartitioner()
               or setupInputs() is failing.
               This function may be called multiple times; the error
               strings are appended.
     requires  arg non-null.
  */

public:
  APT_ErrorLog& errorLog();
  /*
     effect    Returns access to this partitioner's internal error log
               object.
     note      Any traffic generated on this log will be identified as
               originating from this partitioner.
  */


  /**** functions called from setupInputs() overrides ****/

  /* This function should only be called from within
     setupInputs().  It should never be called from within
     describePartitioner() or partitionInput(). */

protected:
  void setupInputAccessor(const APT_FieldSelector& component,
                          APT_InputAccessorBase* acc);
  /*
     effect    Attaches the indicated accessor object to the
               indicated component of the input records.
               The component argument is evaluated relative to the
               schema that was specified in the call to
               setInputInterfaceSchema().
     requires  acc non-null
               The component must not already have an accessor.
               If acc->hasType() is true, then component must identify
               a field component of the indicated dataset's interface
               schema whose type exactly matches the accessor's
               datatype.
```

```
                If acc->hasType() is false, then the accessor's type
                will be determined by the field component.
  */

  APT_InputInterface* inputInterface();
  /*
     effect     Returns this partitioner's input interface object.
  */

public:
  virtual void setOperator(APT_Operator* op);
  virtual void activate();
  // for DB2 partitioner; what to do with this long-term is TBD

private:
  friend class APT_SC; // access to setOutputPartitions()
  friend class APT_OperatorRep;
  friend class APT_DataSet;
  friend class APT_DataSetRep;
  friend class APT_StepRep;
  friend class APT_PartitionerRep;
  friend APT_Archive& operator|| (APT_Archive&, APT_DataSetRep&);

  // prohibit copying
  APT_Partitioner(const APT_Partitioner&);
  APT_Partitioner& operator= (const APT_Partitioner&);

  const APT_PartitionerSC* scInterface() const;
  APT_PartitionerSC* scInterface();
  /*
     effect     Returns a pointer to an object providing supplemental
                interface to this dataset.  For the step compiler
                implementation.
     note       The caller must not delete the returned pointer.
  */

  APT_PartitionerRep* rep_;
};

#endif // APT_PARTITIONER_H
```

```
// -*-Mode: C++-*-
//
// Copyright (C) 1996 Torrent Systems, Inc. All Rights Reserved.
//
// $Id: pipecon.h,v 1.13 2000/02/17 16:20:09 smr Exp $

#ifndef APT_PIPECON_H
#define APT_PIPECON_H

#ifndef APT_GSUBPROC_H
#include <apt_framework/gsubproc.h>
#endif

class APT_PipeConnectionImpl;
class APT_PipeConnection: public APT_GeneralSubprocessConnection
{
  public:

    APT_PipeConnection();
    ~APT_PipeConnection();

    APT_PipeConnection(Direction direction,
                       APT_UInt32 fileDescriptor);
    /*
       effects  Create a pipe-based connection with the specified
                direction, accessed via the specified file descriptor
                in the subprocess.
    */


    virtual APT_String identifier() const;
    /*
       effects  Returns "fd <#>"
    */



    virtual int localFileDescriptor();
    /*
       effects  Return the file descriptor associated with the
                "local" end of the connection.

       requires This may only be called from runConnection().
    */

    int remoteFileDescriptor() const;
    /*
       effects Return the file descriptor the pipe is suppose
               use as the remote end of the connection.
    */

  protected:

    virtual APT_Status
      reportConnectionAttachment(const APT_GeneralSubprocessConnection * connection);
```

```
    virtual APT_Status create(const APT_GeneralSubprocessOperator * op);
    virtual APT_Status disableConnection();
    virtual APT_Status enableConnection();
    virtual APT_Status setupInSubprocess(APT_GeneralSubprocessOperator * op);
    virtual APT_Status terminateConnection();
    virtual bool supportsFileSets() const;
    virtual bool supportsDirectFiles() const;

    virtual void setupBuffers();

    APT_PipeConnectionImpl * pImpl_;

  private:

    static APT_PipeConnectionImpl * pFirst_;
    static APT_PipeConnectionImpl * pLast_;

    APT_DECLARE_RTTI(APT_PipeConnection);
    APT_DECLARE_PERSISTENT(APT_PipeConnection);
};


class APT_NullConnection : public APT_PipeConnection
{
  public:
    APT_NullConnection();
    ~APT_NullConnection();

    APT_NullConnection(Direction direction,
                       APT_UInt32 fileDescriptor);
    /*
       effects  Create a pipe-based connection with the specified
                direction, accessed via the specified file descriptor
                in the subprocess. The file descriptor will actually
                be connected to /dev/null, so a "source" connection
                will read as end-of-file, and a "sink" connection will
                silently discard data.
    */

  protected:

    // Hide and disable these routines.

    virtual int localFileDescriptor();
    virtual bool supportsFileSets() const;
    virtual bool supportsDirectFiles() const;

    // Override to hook descriptor to /dev/null.
    APT_Status setupInSubprocess(APT_GeneralSubprocessOperator * op);

  private:

    APT_DECLARE_RTTI(APT_NullConnection);
    APT_DECLARE_PERSISTENT(APT_NullConnection);
};
```

```
#endif // APT_PIPECON_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_RAWFIELD_H
#define APT_RAWFIELD_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_ASSERT_H
#include <apt_util/assert.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

#include <string.h>

class ostream;
class istream;

class APT_RawField
/* Class representing a raw field of a dataset record.

   An APT_RawField has the following public characteristics:

     - can be fixed-length or variable-length

   No derivations are anticipated.

   TBD: update to reflect alignment
*/
{
public:
  APT_RawField();
  // defaults to variable length, empty

  APT_RawField(const APT_RawField&);
  // assignment is below

  ~APT_RawField();
```

```
void setFixedLength(APT_UInt32 len);
// requires: len > 0

bool isFixedLength() const { return fixedLength_; }
/*
   effect    Tells if this raw field is fixed length.  If so,
             length() will always report the same value.
*/

void setBoundedLength(APT_UInt32 len);
// requires: len > 0

void setVariableLength();

bool isBoundedLength() const { return isBoundedLength_; }
/*
   effect    Tells if this raw field is bounded length.
*/

bool isVariableLength() const { return !fixedLength_; }
/*
   effect    Tells if this raw is variable-length.
*/

APT_UInt32 length() const
{ if (!isBoundedLength_) return length_;
  else return (length_ > boundedLength_) ? boundedLength_ : length_; }
/*
   effect    Returns the length of this raw field.
*/

void setLength(APT_UInt32 len);
/*
   effect    Sets the length of this raw field.
             If the length is reduced, the raw's right-most
             bytes are dropped; if the length is increased, the
             raw is right-filled with 0-valued bytes.
             If isBoundedLength() is true, silently clamps the argument
             length according to the bound.
   requires  isVariableLength() must be true.
             0 <= len <= 0x7fffffff
             Must not be called on an input raw field.
*/

void* content() { return *basePtr_ + offset_; }
const void* content() const { return *basePtr_ + offset_; }
/*
   effect    Returns a pointer to this raw field's contents.
             Returns 0 if this raw field's length() is zero.
```

```
   note      The caller may access up to length() bytes using the
             returned pointer.
             A raw field's content() can have any address alignment.
             The caller must not retain the returned pointer, as the
             underlying storage can be invalidated by many things,
             including (but not least) non-const operations on this
             raw field.
*/

APT_RawField& operator= (const APT_RawField& rhs);
/*
   effect    If isFixedLength() is true, copies min(length(),
             rhs.length()) bytes from rhs into this raw field's
             content.  If less than length() bytes are copied, the
             balance of the length() is right-filled with 0-valued
             bytes.
             If isBoundedLength() is true, silently clamps the copied
             length according to the bound.
             This possibly reallocates storage for this raw, and
             copies all of rhs's content into this raw.
   note      For variable-length raw fields, it is not necessary to
             call setLength() before calling this function.
   requires  Must not be called on an input raw field.
*/

void assignFrom(const void* content, APT_UInt32 len);
/*
   effect    Like operator=, except for the form of the data source.
   requires  0 <= len <= 0x7fffffff
             Must not be called on an input raw field.
*/

void bind(const void* content, APT_UInt32 len);
/*
   effect    Like assignFrom(), except that under certain
             circumstances the data are not copied and are instead
             referenced.
   requires  0 <= len <= 0x7fffffff
             The content must remain valid until either this raw
             field is destroyed, or a bind() or assignFrom() is
             performed.
             Must not be called on an input raw field.
*/

// equality comparison based on content
friend bool
  operator== (const APT_RawField& lhs, const APT_RawField& rhs);
friend bool
  operator!= (const APT_RawField& lhs, const APT_RawField& rhs)
```

```
    { return !(lhs == rhs); }

  friend ostream& operator<< (ostream&, const APT_RawField&);
  /*
     effect    Prints all length() bytes of this raw field, using
               syntax:
                 {xx xx xx}
               Where each xx is either a pair of hex digits or a
               single printable ASCII character.  A hex digit pair
               is used to represent non-printable characters.
               A character or hex digit pair is printed for each byte
               of the raw field.  If the raw field is zero-length,
               then {} is printed.
  */
  friend istream& operator>> (istream&, APT_RawField&);
  /*
     effect    Extracts a raw field, expecting the format printed by
               the ostream inserter (operator<<).
               Leading whitespace, if any, is skipped.
  */

  void clear();
  // make this raw empty (if var-len) or padded (if fixed-len)

private:
  friend class APT_RawFieldDescriptor;
  friend class APT_RawField_UT;

  void allocBuf(APT_UInt32 newSize, const char* src, APT_UInt32 srcLen);

  void prepareForFielding(bool isOutput);
  /* prepares this instance to be managed as a field value by the
     framework */

  void becomeEmpty()
  {
    delete[] ourAlloc_;
    ourAlloc_ = 0;
    allocLength_ = 0;
    base_ = 0;
    length_ = 0;
  }
  bool equals(const void*, APT_UInt32 len) const;

  bool isField_;               // is this a field managed by the framework?
  bool isOutput_;              // valid iff isField_ is true

  bool fixedLength_;
```

```
  bool isBoundedLength_;
  APT_UInt32 boundedLength_;

  char* const* basePtr_;
  APT_UInt32 offset_;

  char* base_;
  APT_UInt32 length_;

  char* ourAlloc_;              /* points to storage we allocated (can
                                   be different than base_ if bind() or
                                   direct protocol set occurs) */
  APT_UInt32 allocLength_;

  APT_DECLARE_NEW_AND_DELETE(APT_RawField);
};

inline void APT_RawField::bind(const void* c, APT_UInt32 len)
{
  if (isField_) APT_ASSERT_DEBUGONLY(isOutput_);

  if (!c) len = 0;

  if (fixedLength_)
  {
    APT_UInt32 xfer = length_;
    if (len < xfer) xfer = len;

    if (xfer) memcpy(content(), c, xfer);
    if (length_ != xfer) memset((char*) content()+xfer, 0, length_-xfer);
  }
  else  // variable length
  {
    if (isBoundedLength_ && len > boundedLength_)
      len = boundedLength_;

    base_ = (char*) c;
    length_ = len;
  }
}

inline void APT_RawField::clear()
{
  if (fixedLength_)
    memset(content(), 0, length_);
  else  // variable length
  {
    // leave allocation as-is
    base_ = 0;
```

```
    length_ = 0;
  }
}

#endif // APT_RAWFIELD_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_SCHEMA_H
#define APT_SCHEMA_H

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_PROPLIST_H
#include <apt_util/proplist.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif

class APT_Lexer;
class APT_ParseError;
class APT_ErrorLog;
class APT_FieldSelector;
class APT_Schema;
class APT_SchemaField;
class APT_SchemaTypeSpec;
class APT_SchemaAggregate;
class APT_SchemaFieldListRep;
class APT_ModifyAdapter;
class APT_FieldTypeDescriptor;
class APT_RawFieldDescriptor;
class APT_Identifier;
```

```
class istream;


/* Dataset record schema API. */

class APT_SchemaFieldList
/* Common base class for APT_Schema and
   APT_SchemaAggregate, to facilitate clients who want to
   treat each kind of field list in a similar fashion.

   In other words, this class is an implementation hack.
*/
{
protected:
  APT_SchemaFieldList();
  APT_SchemaFieldList(const APT_SchemaFieldList&);
  APT_SchemaFieldList& operator= (const APT_SchemaFieldList&);
  ~APT_SchemaFieldList();

public:
  int numFields() const;
  /*
     effect    Tells how many fields this list has.  0 by default.
  */

  const APT_SchemaField& field(int pos) const;
  APT_SchemaField& field(int pos);
  /*
     effect    Returns the indicated field.
     note      The returned reference is valid only as long as this
               APT_SchemaFieldList is not modified or destroyed.
     requires  0 <= pos < numFields()
  */

  const APT_SchemaField& field(const APT_FieldSelector& select) const;
  APT_SchemaField& field(const APT_FieldSelector& select);
  const APT_SchemaField& field_byId(const APT_Identifier& id) const;
  APT_SchemaField& field_byId(const APT_Identifier& id);
  /*
     effect    Returns the indicated field.
               Matching is case-independent.
     note      The returned reference is valid only as long as this
               APT_SchemaFieldList is not modified or destroyed.
               The _byId forms search within this scope; the fieldsel
               forms can descend into sub-scopes.
     requires  select.isSubscripted() is false.
               hasField(select) must be true
  */
```

```
bool hasField(const APT_FieldSelector& select) const;
bool hasField_byId(const APT_Identifier& id) const;
/*
   effect    Tells if the indicated selector refers to a field of
             this list.
             Matching is case-independent.
   note      The _byId form searches within this scope; the fieldsel
             form can descend into sub-scopes.
   requires  select.isSubscripted() is false.
*/

const APT_SchemaField* lookup(const APT_FieldSelector& select) const;
APT_SchemaField* lookup(const APT_FieldSelector& select);
const APT_SchemaField* lookup_byId(const APT_Identifier& id) const;
APT_SchemaField* lookup_byId(const APT_Identifier& id);
/*
   effect    A combination of hasField() and field(); returns null
             if not found.
   note      The _byId forms search within this scope; the fieldsel
             forms can descend into sub-scopes.
*/

int addField(const APT_SchemaField& f, int pos=-1);
/*
   effect    Adds a copy of the indicated field to this
             list.  The pos argument specifies what position the
             field is to occupy; a pos value of -1 means add
             the field to the end (this is most efficient).
   note      Other fields are moved (without reordering) to
             accommodate the new field.
   returns   The position index at which the field was added.
   requires  0 <= pos < numFields()+1  or pos==-1
             f's identifier must not be a duplicate of any other
             field of this list.
*/

int addField_flagDup(const APT_SchemaField& f, bool* dupFlag, int pos=-1);
/*
   effect    Like addField(), except a duplicate field is flagged
             rather than being a requires violation.  In the case of
             a duplicate, this field list is not modified, the
             dupFlag is set to true, and -1 is returned.
*/

void addFields(const APT_SchemaFieldList& other);
/*
   effect    Adds a copy of the argument's fields to this list.
             The fields are added in order, and are added to the
             end of this list.
```

```
   requires  There must be no duplicate among this list's
             field identifiers and the argument list's field
             identifiers.
*/

void removeField(int pos);
/*
   effect    Removes the field at the indicated position.
   note      Other fields are moved (without reordering) to
             close the gap vacated by the removed field.
   requires  0 <= pos < numFields()
*/

bool isConcreteSchema() const;
/*
   effect    Tells if this schema is usable as a concrete schema.  A
             concrete schema contains no schema variables (fields of
             type "*") and no untyped fields.
*/

bool isOutputSchema() const;
/*
   effect    Tells if this schema is usable as an output interface
             schema.  An output interface may have schema variables,
             but may not have any untyped fields.
*/

bool isInterfaceSchema() const;
/*
   effect    Tells if this schema is usable as an interface schema.
             An interface schema may have schema variables (fields
             of type "*").
*/

void removeFieldProperties();
/*
   effect    Removes the property list from all fields (and their
             sub-fields, if subrec or tagged).
*/

bool operator== (const APT_SchemaFieldList&) const;
/*
   effect    Tells if the two schemas are identical in every
             observable way.
   note      Property lists are not compared.
*/
bool operator!= (const APT_SchemaFieldList& rhs) const
{ return !(*this == rhs); }
```

```
  bool isEqualWithProps(const APT_SchemaFieldList&) const;
  /*
     effect    Like operator==, except also takes property lists into
               account.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaFieldList&);

  void repInvariant() const;

  void expandIntact();
  /*
     effect    For all fields of this scope for which isIntactRaw() is
               true, replaces those fields with the described (and
               non-dropped) fields of their intact schema.
     note      For a top-level schema, desugarIntact() should have
               been called before expandIntact().
     For Torrent use only
  */

  bool hasIntact() const;
  // tells if this scope has any fields for which isIntactRaw() is true

protected:
  const APT_SchemaField* owner() const;
  void setOwner(APT_SchemaField*);

  void setDefaultNullable(bool);

  APT_Status checkForIntactNameConflicts(APT_String* info) const;
  /* looks for field name conflicts between top-level fields of
     this scope and top-level fields of any intact records in this
     scope */

private:
  friend class APT_SchemaField;
  friend class APT_RawFieldDescriptor;
  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaField&);

  void cow();
  APT_SchemaFieldListRep* rep_;
};


/**** APT_Schema ****/

class APT_Schema : public APT_SchemaFieldList
{
public:
```

```
APT_Schema();
~APT_Schema();

APT_Schema(const APT_Schema&);
APT_Schema& operator= (const APT_Schema&);

APT_Schema(const char*, APT_ParseError* err=0);
APT_Schema(istream&, APT_ParseError* err=0);
/*
   effect    Parses the input, expecting a record description
             (starting with the token 'record').
             If the parse is successful, this APT_Schema
             object's state is replaced by the parsed record
             description.
   throws    APT_ParseError: trouble parsing the supplied input as a
               record declaration.  If the err argument is 0, then
               the error is thrown; if err is non-null, then the
               error object is copied into the object pointed to by
               err.
   requires  string arg non-null
*/

bool isFixedLength() const;
/*
   effect    Tells if the records of this schema are fixed length.
   DEPRECATED
*/

APT_UInt32 recordLength() const;
/*
   effect    Tells how large this schema's records are when stored
             to a file-based persistent dataset.
   note      The returned value is suspect (the API guts uses
             alignment logic that differes from that employed here).
   requires  isFixedLength() is true.
             isConcreteScema() is true.
   DEPRECATED
*/

APT_PropertyList properties() const;
void setProperties(const APT_PropertyList&);
/*
   effect    Accesses this schema's property list.
             Defaults to empty.
*/

bool isIntact() const;
/*
   effect    Tells if this is an intact schema.  An intact schema
```

```
            has exactly one top-level "intact" property whose value
            is either an identifier or empty.
            Alternately, an intact schema is one without a top-level
            "intact" property, but with exactly one intact raw field.
*/
APT_Identifier intactName() const;
/*
   effect    Returns the name of this intact schema.
   requires  isIntact() must be true.
*/

void parse(const char*, APT_ParseError* err=0);
void parse(istream&, APT_ParseError* err=0);
/*
   effect    Parses the input, expecting a record description
             (starting with the token 'record').
             If the parse is successful, this APT_Schema
             object's state is replaced by the parsed record
             description.
   throws    APT_ParseError: trouble parsing the supplied input as a
               record declaration.  If the err argument is 0, then
               the error is thrown; if err is non-null, then the
               error object is copied into the object pointed to by
               err.
             If an error occurs, this APT_Schema object's
             state is unchanged.
   requires  string arg non-null
*/

void unparse(ostream&, int indent=0) const;
APT_String unparse(int indent=0) const;
/*
   effect    Prints or returns a parse()able representation of
             this APT_Schema.
*/

bool operator== (const APT_Schema&) const;
/*
   effect    Tells if the two schemas are identical in every
             observable way.
   note      Property lists are not compared.
*/
bool operator!= (const APT_Schema& rhs) const
{ return !(*this == rhs); }

bool isEqualWithProps(const APT_Schema&) const;
/*
   effect    Like operator==, except also takes property lists into
             account.
```

```
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_Schema&);

  void repInvariant() const;

  void parse_(APT_Lexer&, APT_ParseError* err);
  // for Torrent use only

  void unparse_nosugar(ostream&, int indent=0) const;
  // for Torrent use only

  void sugarIntact();
  /*
     effect    If this is an intact schema, puts it into the form:

                 record { intact=R, ... } (fields...)

               If isIntact() is false, there is no effect.  If this
               schema is already sugared, there is no effect.
     requires  There must not be any intacts nested within intacts.
     For Torrent use only
  */
  void desugarIntact(APT_ErrorLog& log);
  /*
     effect    If this is an intact schema, puts it into the form:

                 record { ... } (R:raw[record { ... } (fields...)]

               The top-level record properties (both at record-level,
               and in the raw's schema) will not have an 'intact'
               property.
               If isIntact() is false, there is no effect.  If this
               schema is already desugared, there is no effect.
     requires  There must not be any intacts nested within intacts.
     errors    If there is trouble setting the raw's schema (because
               the schema is not a valid import schema, for example),
               error information is written to the supplied log.
     For Torrent use only
  */

  bool isInterfaceSchema() const;
  // adjusts the inherited method; see inherited description

private:
  APT_PropertyList properties_;
};
```

```
/**** APT_SchemaAggregate ****/

class APT_SchemaAggregate : public APT_SchemaFieldList
{
public:
  APT_SchemaAggregate();        // defaults to eSubrec, no fields
  ~APT_SchemaAggregate();
  APT_SchemaAggregate(const APT_SchemaAggregate&);
  APT_SchemaAggregate& operator= (const APT_SchemaAggregate&);

  enum Kind
  {
    eSubrec,                    /* like a C struct */
    eTagged                     /* discriminated union. */
  };
  Kind kind() const;
  void setKind(Kind);
  /*
     effect    Identifies the kind of aggregate this is.
               Defaults to eSubrec.
  */

  bool isFixedLength() const;
  /*
     effect    If this is a subrec, tells if all of its fields
               are fixed length.
               If this is a tagged, returns false.
     DEPRECATED
  */

  APT_UInt32 aggrLength() const;
  /*
     effect    Tells how large this aggregate is when stored to a
               file-based persistent dataset.
     note      The returned value is suspect (the API guts uses
               alignment logic that differes from that employed here).
     requires  isFixedLength() is true.
               Must not contain any schema variable fields
     DEPRECATED
  */

  bool operator== (const APT_SchemaAggregate&) const;
  /*
     effect    Tells if the two objects are identical in every
               observable way.
     note      Property lists are not compared.
  */
  bool operator!= (const APT_SchemaAggregate& rhs) const
  { return !(*this == rhs); }
```

```
  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaAggregate&);

  void repInvariant() const;

private:
  friend class APT_Schema;
  friend class APT_SchemaField;
  friend class APT_SchemaTypeSpec;
  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaField&);

  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
     effect    Parses the input, expecting an aggregate description
               (starting with the token 'subrec' or 'tagged').
               The record schema object being constructed is passed
               as sch.
               This Aggregate object's state is replaced by the
               parsed record description.
               The supplied error object is set accordingly if parse
               error(s) occurs.
     requires  err non-null
  */

  void unparse_(ostream&, int indent, const char* props,
                bool doSugar=true) const;
  /*
     effect    Prints a parse()able representation of this Aggregate.
  */

  Kind kind_;
};  // APT_SchemaAggregate


/**** APT_SchemaTypeSpec ****/

class APT_SchemaTypeSpec
/* This class is deprecated.  The new way to describe a schema type
   is via a class derived from APT_FieldTypeDescriptor (see
   apt_framework/type/descriptor.h).
*/
{
private:
  /* For support of existing APT_SchemaTypeSpec-based code (and
     persistent archives containing serialized representations of
     typespecs), we provide for conversions between APT_SchemaTypeSpec
     and corresponding derived APT_FieldTypeDescriptor instances.
  */
  const APT_FieldTypeDescriptor* getDescriptor() const;
```

```
  /*
     effect    Retrieves a type descriptor corresponding to this
               typespec.
     requires  type() must not be eAggregate or eVar.
  */
  void setFromDescriptor(const APT_FieldTypeDescriptor*);
  /*
     effect    Sets this typespec according to the given type
               descriptor.
     requires  arg non-null.
               arg must correspond to a "v1.1" type.
  */

public:
  /* Everything below is deprecated.  However, we don't bother putting
     "deprecated" messages into all of them; instead, the typespec
     getter/setter functions on APT_SchemaField issue "deprecated"
     messages. */

  APT_SchemaTypeSpec();         // defaults to int32
  APT_SchemaTypeSpec(const APT_SchemaTypeSpec&);
  APT_SchemaTypeSpec& operator= (const APT_SchemaTypeSpec&);
  ~APT_SchemaTypeSpec();

  APT_SchemaTypeSpec(const char*, APT_ParseError* err=0);
  APT_SchemaTypeSpec(istream&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a type specification
               (without a ':' at the beginning).
               If the parse is successful, this TypeSpec object's
               state is replaced by the parsed type specification.
     throws    APT_ParseError: trouble parsing the supplied input as a
               type declaration.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
     requires  string arg non-null
  */

  enum TypeCode
  {
    eInt=0,                     /* integer (8, 16, or 32 bits) */
    eFloat,                     /* float or double */
    eString,                    /* fixed-length or encoded-length
                                   character string */
    eRaw,                       /* fixed-length or encoded-length
                                   uninterpreted byte array */
    eUser,                      /* user-defined type */
    eAggregate,                 /* subrec/tagged type */
```

```
  eVar                        /* schema variable (for interface
                                 schema only) */
};
TypeCode type() const;
void setType(TypeCode);
/*
   effect    Identifies the kind of type spec this is.
             Defaults to eInt (4 bytes, signed).
*/

int accessorType() const;
/*
   effect    Returns an integer value corresponding to the
             APT_AccessorBase::Type enum.  The returned value
             identifies the kind of accessor that should be used to
             access a field of this type.
   requires  type() != eAggregate
             type() != eVar
*/

bool isFixedLength() const;
/*
   effect    Tells if this is a fixed-length type.
   requires  type() != eAggregate
             type() != eVar
*/

APT_UInt32 typeLength() const;
/*
   effect    Tells how large this type is when stored to a
             file-based persistent dataset.
   requires  isFixedLength() is true.
             type() != eAggregate
             type() != eVar
*/

APT_PropertyList properties() const;
void setProperties(const APT_PropertyList&);
/*
   effect    Accesses this type's property list.
             Defaults to empty.
*/

bool operator== (const APT_SchemaTypeSpec&) const;
/*
   effect    Tells if the two objects are identical in every
             observable way.
   note      Property lists are not compared.
*/
```

```
bool operator!= (const APT_SchemaTypeSpec& rhs) const
{ return !(*this == rhs); }

APT_String unparse() const;
/*
   effect    Returns a parseable representation of this type spec.
*/

friend APT_Archive& operator|| (APT_Archive&, APT_SchemaTypeSpec&);

void repInvariant() const;

/**** stuff for eInt ****/

/* type() must be eInt before setting any integer attributes */

int intNumBits() const;       // defaults to 32
void setIntNumBits(int);      // requires: 8, 16, or 32

bool isUnsigned() const;      // defaults to false
void setIsUnsigned(bool);

/**** stuff for eFloat ****/

/* type() must be eFloat before setting any floating point attributes */

bool isDouble() const;
void setIsDouble(bool);       // defaults to false

/**** stuff for eString ****/
/**** stuff for eRaw ****/

/* type() must be eString or eRaw before setting any of these */

enum LengthType { eVariableLength=0, eFixedLength };
LengthType lengthType() const;
void setLengthType(LengthType);
/*
   effect    Accesses whether the string (or raw) is fixed or variable
             length.
             Defaults to eVariableLength.
*/

APT_UInt32 fixedLength() const;
void setFixedLength(APT_UInt32);
/*
   effect    Accesses the fixed length of this string (or raw).
   requires  lengthType() == eFixedLength
             value must be > 0
```

```
  */

  /* stuff for raw only */
  /* type() must be eRaw before setting this */

  bool isAligned() const;
  void setIsAligned(bool);      // defaults to false


  /**** stuff for eAggregate ****/
  // see APT_SchemaField::aggregate()

  /**** stuff for eVar ****/
  // none

private:
  friend class APT_Schema;
  friend class APT_SchemaField;
  friend class APT_SchemaLengthSpec;
  friend class APT_ModifyAdapter;
  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaField&);

  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
     effect    Parses the input, expecting a type description
               (with the token ':' having been consumed already).
               This TypeSpec object's state is replaced by the
               parsed type spec description.
               The supplied error object is set accordingly if parse
               error(s) occurs.
     requires  err non-null
  */

  void unparse_(ostream&, int indent) const;
  /*
     effect    Prints a parse()able representation of this TypeSpec.
  */

  int typeAlignment() const;
  /*
     effect    Tells what alignment this type must have if it is to
               be accessed via a native pointer.
     note      Returns 1, 2, 4, or 8.  Platform dependent.
     requires  type() != eAggregate
               type() != eVar
  */

  APT_UInt32 stateWord_;
  APT_PropertyList properties_;
```

```
  union
  {
    APT_String* userType_;
    APT_UInt32 fixedLength_;
    APT_SchemaAggregate* aggregate_;
  };
};  // APT_SchemaTypeSpec


/**** APT_SchemaField ****/

class APT_SchemaField
{
public:
  APT_SchemaField();
  // default is scalar value of type int32

  APT_SchemaField(const APT_SchemaField&);
  APT_SchemaField& operator= (const APT_SchemaField&);
  ~APT_SchemaField();

  const APT_Identifier& ident() const { return identifier_; }
  void setIdentifier(const char* id);
  /*
     effect    Specifies the identifier of this field.  A
               default-constructed Field object's identifier() is
               the empty string.
     requires  id must be a legal identifier.
               id must be non-null.
               This field must not currently be a part of a schema or
               aggregate.
  */
  APT_String identifier() const // for back-compatibility
    { return identifier_.string(); }

  APT_FieldSelector path() const;
  void path(APT_FieldSelector*) const;
  /*
     effect    Returns the fully-qualified name of this field.  A
               top-level field's path() is the same as its
               identifier().
               When a field is part of an aggregate, its path()
               reflects its nesting within the record.
     note      The second form is a bit faster
  */

  int pos() const;
  /*
     effect    Returns the position of this field within the record (or
```

```
            aggregate).  -1 means the field does not have a position
            (because it is not part of a record or aggregate).
*/

enum Occurrences { eScalar, eFixedLengthVector, eVariableLengthVector };
Occurrences occurrences() const;
void setOccurrences(Occurrences);
/*
   effect    Accesses the kind of field: scalar (single value),
             fixed-length vector, or variable-length vector.
             Defaults to eScalar.
   requires  If this field is a schema variable, must be eScalar.
*/

APT_UInt32 fixedVectorLength() const;
void setFixedVectorLength(APT_UInt32);
/*
   effect    Accesses the fixed vector length.
   requires  occurrences() == eFixedLengthVector
             value must be > 0
*/

bool isFixedLength() const;
/*
   effect    Tells if this field is fixed length (scalar or fixed-length
             vector of fixed-size type).
   DEPRECATED
*/

bool isFixedLength_elt() const;
/*
   effect    Tells if this field's type is fixed length (regardless
             of whether this field is a varible-length vector).
   DEPRECATED
*/

APT_UInt32 fieldLength() const;
/*
   effect    Tells how large this field is when stored to a
             persistent dataset.
   requires  isFixedLength() is true.
             field must not be a schema variable
   DEPRECATED
*/

int fieldAlignment() const;
/*
   effect    Tells what alignment this field must have if it is to
             be accessed via a native pointer.
```

```
   note      Returns 1, 2, 4, or 8.  Platform dependent.
   requires  field must not be a schema variable
   DEPRECATED
*/

enum Kind { eValue, eSchemaVariable, eAggregate };
Kind kind() const { return kind_; }
void setKind(Kind);
/*
   effect    Determines what kind of field this is.
*/

bool hasType() const;
/*
   effect    Tells if this field has a type.  Default is true.
   requires  kind() must be eValue.
*/
void setUntyped();
/*
   effect    Causes hasType() to become false.
   requires  kind() must be eValue.
*/

const APT_FieldTypeDescriptor* typeDescriptor() const;
/*
   effect    Returns an object that describes this field's type.
   note      The returned descriptor is a registered copy.
   requires  kind() must be eValue.
             hasType() must be true.
*/

void setTypeDescriptor(const APT_FieldTypeDescriptor*);
/*
   effect    Sets this field's type.
   note      A copy of the argument is made via registeredCopy().
   requires  arg non-null
             kind() must be eValue.
*/

void setTypeDescriptor(const char*, APT_ParseError* err=0);
/*
   effect    Sets this field's type from the supplied, possible
             parameterized, schema type.
   throws    APT_ParseError: the indicated schema field type could
               not be found, or the supplied parameters could not
               be parsed.  If the err argument is 0, then the error
               is thrown; if err is non-null, then the error object
               is copied into the object pointed to by err.
   requires  string arg non-null
```

```
              kind() must be eValue.
  */

  const APT_SchemaTypeSpec& typeSpec() const; // requires: hasType() true
//  APT_SchemaTypeSpec& typeSpec();     // its equivalent is not implemented
  void setTypeSpec(const APT_SchemaTypeSpec& ts);
  /*
     effect    Accesses this field's type spec.
     note      The returned reference is valid as long as this field is
               not modified or destroyed.
     DEPRECATED
  */

  bool isNullable() const { return nullable_; }
  /*
     effect    Tells if this field is nullable.  Default is false.
     requires  kind() must be eValue.
  */
  void setNullable(bool);
  /*
     effect    Sets whether this field is nullable.
     requires  kind() must be eValue.
  */

  APT_PropertyList properties() const;
  void setProperties(const APT_PropertyList&);
  /*
     effect    Accesses this field's property list.
               Defaults to empty.
  */

  const APT_SchemaAggregate& aggregate() const;
  APT_SchemaAggregate& aggregate();
  void setAggregate(const APT_SchemaAggregate&);
  /*
     effect    Accesses the aggregate object containing this field's
               sub-fields.
     note      The returned reference is valid as long as this field is
               not modified or destroyed.
     requires  kind() == eAggregate
  */

  bool operator== (const APT_SchemaField&) const;
  /*
     effect    Tells if the two objects are identical in every
               observable way.
     note      Field identifiers are compared in a case-independent
               fashion.  The path() is not compared.  Property lists
               are not compared.
```

```
  */
  bool operator!= (const APT_SchemaField& rhs) const
  { return !(*this == rhs); }

  bool isEqualWithProps(const APT_SchemaField&) const;
  /*
     effect    Like operator==, except also takes property lists into
               account.
  */

  bool isIntactRaw() const;
  /*
     effect    Tells if this field is a raw parameterized by an
               intact schema.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaField&);

  void repInvariant() const;

private:
  friend class APT_SchemaFieldList;
  friend class APT_SchemaFieldListRep;
  friend class APT_Schema;
  friend class APT_SchemaAggregate;
  friend APT_Archive& operator|| (APT_Archive&, APT_Schema&);

  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
     effect    Parses the input, expecting a field description
               (starting with an identifier token).
               The parent argument identifies (by full name) the
               parent of this field.
               This Field object's state is replaced by the parsed
               field description.
               The supplied error object is set accordingly if parse
               error(s) occurs.
               If an error occurs, input is consumed until a
               semicolon is seen before returning or throwing.
     requires  err non-null
  */

  void unparse_(ostream&, int indent, bool doSugar=true) const;
  /*
     effect    Prints a parse()able representation of
               this Field.
     requires  identifier() is non-empty.
  */
```

```
  bool hasIntact() const;
  /* tells if this field is an intact raw, or is an aggregate that
     has any fields for which hasIntact() is true */

  void setDefaultNullable(bool);
  // determines nullability_ if nullableSet_ is false

  void setParent(APT_SchemaFieldListRep*);

  APT_Identifier identifier_;
  APT_SchemaFieldListRep* parent_;
  APT_UInt32 fixedVecLen_;
  bool nullable_;
  bool nullableSet_;            /* aux flag to interact with top-level
                                   nullability spec during record parse */
  APT_PropertyList properties_;
  Kind kind_;
  APT_SchemaAggregate* aggregate_;
  const APT_FieldTypeDescriptor* typeDesc_;
  APT_SchemaTypeSpec* typeSpec_; /* allocated only if needed to support
                                    deprecated function */
};  // APT_SchemaField


// internal class
class APT_SchemaLengthSpec
{
public:
  APT_SchemaLengthSpec();               // defaults to eNone

  // default copy, assign, dtor OK

  enum Kind
  {
    eNone,                      /* indicates field to which this
                                   lengthspec belongs is not a vector */
    eFixed,                     /* field vector length (or
                                   string/raw field length)
                                   is a fixed length */
    eIntPrefix                  /* field vector length (or
                                   string/raw field length)
                                   is variable length, with the length
                                   being given by an implicit integer
                                   field immediately preceding the
                                   variable length component. */
  };
  Kind kind() const;
  void setKind(Kind);
  /*
```

```
    effect    Identifies the kind of length spec this is.
              Defaults to eNone.
  */

  bool isFixedLength() const;
  /*
    effect    Tells if this length spec is fixed length.
  */

  APT_UInt32 fixedLength() const;
  void setFixedLength(APT_UInt32);
  /*
    effect    Specifies the fixed length value.
    requires  kind() must be eFixed.
              value must be > 0
              TBD: allow 0 length value?
  */

  bool operator== (const APT_SchemaLengthSpec&) const;
  /*
    effect    Tells if the two objects are identical in every
              observable way.
  */
  bool operator!= (const APT_SchemaLengthSpec& rhs) const
  { return !(*this == rhs); }

  friend APT_Archive& operator|| (APT_Archive&, APT_SchemaLengthSpec&);

  void repInvariant() const;

private:
  friend class APT_Schema;
  friend class APT_SchemaField;
  friend class APT_SchemaTypeSpec;

  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
    effect    Parses the input, expecting a length spec description
              (starting with the token '[').
              This LengthSpec object's state is replaced by the
              parsed length spec description.
              The supplied error object is set accordingly if parse
              error(s) occurs.
    requires  err non-null
  */

  void unparse_(ostream&, int indent) const;
  /*
    effect    Prints a parse()able representation of this LengthSpec.
```

```
  */

  Kind kind_;
  APT_UInt32 fixedLength_;
};  // APT_SchemaLengthSpec

APT_DIRECTIONAL_SERIALIZATION(APT_Schema);
APT_DIRECTIONAL_SERIALIZATION(APT_SchemaField);
APT_DIRECTIONAL_SERIALIZATION(APT_SchemaTypeSpec);
APT_DIRECTIONAL_SERIALIZATION(APT_SchemaAggregate);

#endif // APT_SCHEMA_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_STEP_H
#define APT_STEP_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_PROPLIST_H
#include <apt_util/proplist.h>  // for APT_PropertyList
#endif


class APT_Operator;
class APT_DataSet;
class APT_ErrorLog;

class APT_StepRep;
class APT_StepSC;

/* A step is the unit of execution for ORCHESTRATE.

   A step is built up by attaching one or more operators to the step.
   The operators, together with their attached datasets and embedded
   partitioners, comprise the step.

   When a step is run(), a check operation is performed.  If desired,
   the caller may explicitly call check() before calling run().  The
   following conditions are checked before a step is run:

   - operators have all required datasets attached.
   - both ends of virtual datasets are connected to operators.
   - operators' interface schemas have been properly adapted to
```

```
     dataset schemas.
   - the graph formed by operators and datasets is acyclic.
   - the graph formed by operators and datasets is a tree (TBR).
   - operators' checkConfig() are OK.

   Once a step has been check()ed, it can be run().  Running a step
   involves running the operators on various nodes of the parallel
   machine and arranging for records of dataset partitions to flow
   among the operators; this is the heart of ORCHESTRATE.

   If all operators run to completion, the the step's run() call
   reports success.  However, if any operator fails, the step's run()
   call reports a failure.
*/

class APT_Step : public APT_Persistent
{
  APT_DECLARE_RTTI(APT_Step);
  APT_DECLARE_PERSISTENT(APT_Step);

public:
  APT_Step();
  /*
     effect    This step in constructed in the eOpen state, with no
               attached operators.
  */
  ~APT_Step();
  /*
     effect    Attached operators (if any) are detached.  Detached
               operators are deleted if they were dynamically
               allocated.
  */

  void attachOperator(APT_Operator* op);
  /*
     effect    Attaches the operator to this step.  An operator must
               be attached to a step in order to be run.
     note      If op is dynamically allocated, it becomes owned by
               this step and will be deleted when this step is
               destroyed or reset().
     requires  op non-null
               state() == eOpen
               The operator must not already be attached to this or
               any other step.
  */

  void addNodeConstraint(const char* nodePool);
  void addResourceConstraint(const char* resKind, const char* resPool="");
```

```
/*
   effect    Calls the corresponding function for each of this
             step's attached operators.  See APT_Operator for a
             description of these functions.
   requires  Args non-null.
             This step's operators must already be attached.
*/

enum BufferingPolicy
{
  eInheritBuffering,   /* use $APT_BUFFERING_POLICY */
  eAutomaticBuffering, /* buffer to prevent deadlocks */
  eForceBuffering,     /* buffer all virtual datasets */
  eNoBuffering         /* buffer no virtual dataset */
};

void setBufferingPolicy(BufferingPolicy bp);
BufferingPolicy bufferingPolicy() const;
/*
   effect    Sets/gets the default buffering policy for virtual
             datasets in this step.  The default buffering policy
             can be overridden for a specific dataset by calling
             APT_DataSet::setBufferingParameters().  Initially, the
             buffering policy is eInheritBuffering, which means: use
             the policy specified by the environment variable,
             APT_BUFFERING_POLICY, or use eAutomaticBuffering when
             $APT_BUFFERING_POLICY is undefined.
   warning   Inappropriately specifying eNoBuffering could cause a
             dataflow deadlock during step execution.
*/

void setBufferingParameters(const APT_PropertyList& pl);
APT_PropertyList bufferingParameters() const;
/*
   effect    Sets/gets the default buffering parameters for this step.
             These parameters override those specified by environment
             variables.  They apply to all virtual datasets that are
             buffered.  Calling APT_DataSet::setBufferingParameters()
             overrides these parameters.  Overriding is parameter by
             parameter.  The property list is initially empty.
   warning   Setting the buffering parameters inappropriately could
             cause a dataflow deadlock during step execution.
*/

enum State
{
  eOpen,                     /* step may be modified (operators can
                                be attached) */
```

```
  eCheckFailed,              /* step failed its check() with errors */
  eChecked,                  /* step has been check()ed but not yet
                                run() */
  eRun                       /* step has been run() */
};

State state() const;
/*
   effect    Tells what state this step is in.
*/

void reset();
/*
   effect    Puts this step back to the eOpen state.  Attached
             operators (if any) are detached.  Detached
             operators are deleted if they were dynamically
             allocated.
*/

APT_Status check();
/*
   effect    Statically checks step for errors and compiles the step
             for execution if there are no errors.
             The state() becomes eChecked or eCheckFailed.
             Returns APT_StatusOk if there are no errors in the
             compilation; returns APT_StatusFailed if any errors
             are detected.
   requires  state() must be eOpen.
*/

APT_Status run();
/*
   effect    If step has not been checked, calls check(), and if it
             returns an OK status then it executes the step.
             If step was already checked then it just executes the
             step.
             If successful, the state() becomes eRun.  Otherwise,
             the state becomes eCheckFailed.
             Returns APT_StatusFailed if check() fails (or failed).
             If check() is (was) OK, then returns execution status
             code which is APT_StatusOk if the step succeeded, or
             APT_StatusFailed if any operator of the step failed, or
             if the step has been stopped at a checkpoint (see
             checkpointStatus()).
   requires  state() must not be eRun.
*/

// checkpoint support
```

```
APT_Status setupCheckpointing(const char* checkpointParentDir,
                              APT_ErrorLog*, int nSegments=-1);
/*
   effect    Specifies that this step is to be run in checkpointed
             mode.  In this mode, the step is executed in segments.
             Segments execute one by one in order until one fails or
             the last one finishes.
             Information is saved at each segment boundary to allow
             a segment to be restarted if it fails.  See the
             resume() function.
             The specified checkpointParentDir is used to store a
             subdirectory for this step (named by the current jobid;
             see APT_ErrorConfiguration::get().jobIdNumber()).
             If checkpointParentDir is null or empty, then the
             environment variable APT_CHECKPOINT_DIR is used.  If
             there is no APT_CHECKPOINT_DIR, then the current
             directory is used and an info message issued.
             When this step completes, or is aborted or abandoned,
             the jobid subdirectory is deleted.
             The nSegments parameter can be used to limit the number
             of segments executed by run() and resume() for this
             step.
             If nSegments is zero, run()ning the step causes the
             checkpoint subdirectory to be written but no segments
             are run (resume() can then be used to execute the
             step one segment at a time, as if nSegments was set to
             1).
             If nSegments is negative, all segments are run until
             the step completes or a failure occurs.
   returns   APT_StatusOk: step will be run in checkpointed mode
             APT_StatusFailed: step will not be run in checkpointed
               mode.  This is most likely because:
               - the indicated checkpointParentDir does not exist
               - the indicated checkpointParentDir cannot be written
               - the jobid subdirectory already exists
               The failure reason is written to the optional log
               object.
               After a failure return from this function, this step
               is not checkpointed.
   requires  checkpointParentDir or APT_CHECKPOINT_DIR must be
             non-empty strings
             This function may be called only by the main program.
             The state of this step must be eOpen.
             Once this function is successfully called, it may not
             be called again.
*/
```

```
struct CheckpointInfo
{
  int currentlyOpenSegment_;
  // more info will be provided in future releases...

  CheckpointInfo();
  friend APT_Archive& operator|| (APT_Archive&, CheckpointInfo&);
};

static bool hasCheckpoint(const char* checkpointParentDir,
                          CheckpointInfo* progress, APT_ErrorLog*);
/*
   effect    Tells if the given checkpointParentDir contains a
             valid checkpoint subdirectory for the current jobid.
             If so, progress information is written to the optional
             progress arg.
             If not, reason(s) are written to the optional log as
             errors.
   note      May safely be called while the step is being executed
             in this or another process.
             The jobid must have been set as desired before calling
             this function.
   requires  checkpointParentDir must be non-null
*/

bool isResumable() const;
/*
   effect    For checkpointed steps, this function can be used to
             tell whether the step can be resumed when run() returns
              APT_StatusFailed.
             It returns false if run() returned APT_StatusOk.  It
              always returns false for non-checkpointed steps.
*/

static APT_Status resume(const char* checkpointParentDir,
                         bool *resumable, APT_ErrorLog*);
/*
   effect    Resumes execution of a step that has failed to run to
             completion.
             The run is resumed by restarting the segment following
             the last one that completed successfully.
             The specified checkpointParentDir is expected to
             contain a subdirectory for the step to be resumed
             (named by the current jobid; see
             APT_ErrorConfiguration::get().jobIdNumber()).  When the
             resumed step completes, or is aborted or abandoned, its
             jobid subdirectory is deleted.
             The optional resumable argument is updated in the
```

```
              manner of the isResumable() function.
   note       The jobid must have been set as desired before calling
              this function.
   requires   checkpointParentDir must be non-null
              This process must be able to create and destroy files
              in the jobid subdirectory, and must be able to delete
              the jobid subdirectory.
              The step being resumed must not be running.
              This function must be called on the node where run()
              was originally called.
*/

static void abandon(const char* checkpointParentDir, APT_ErrorLog*);
/*
   effect     Abandons execution of a step that is stopped at a
              checkpoint.
              This involves running a cleanup segment, rolling back
              output persistent data sets to their pre-step state,
              and deleting the jobid checkpoint subdirectory.
   note       The jobid must have been set as desired before calling
              this function.
   requires   This process must be able to destroy files in the jobid
              subdirectory, and must be able to delete the jobid
              subdirectory.
              The step being abandoned must not be running.
              This function must be called on the node where run()
              was originally called.
*/


// grunge...

static void loadedLibrary(const char* libName, const char* className);
/*
   effect     Called on the conductor to note that we have loaded a
              dynamic library.  The libName arg is what
              was passed to APT_LoadLibrary().  The className
              identifies the reason the library was loaded.
              On the players, the step will attempt to re-load these
              libraries.
   requires   This function must be called before the step is
              serialized.
              The libName arg must be non-null.
*/

class LoadedLibraryTracker : public APT_Persistent
{
  APT_DECLARE_RTTI(LoadedLibraryTracker);
```

```
    APT_DECLARE_PERSISTENT(LoadedLibraryTracker);
  public:

    LoadedLibraryTracker();
    ~LoadedLibraryTracker();

    struct LL
    {
      APT_String libName_;
      APT_String className_;

      friend APT_Archive& operator|| (APT_Archive&, LL&);
    };
    int numLL_;
    LL* ll_;

    void loadedLibrary(const char* libName, const char* className);

  private:
    // prohibit copy/assign
    LoadedLibraryTracker(const LoadedLibraryTracker&);
    LoadedLibraryTracker& operator= (const LoadedLibraryTracker&);
  };

  static LoadedLibraryTracker* libTracker();

  void dump(ostream& os) const;
  APT_String dumpStr() const;
  /*
     effect     Writes a textual representation of the step on the
                specified output stream or returns it as a string.
                This is pretty thin.  I didn't bother dumping a
                complete description, since this is for debugging
                and there's a LOT of data, and I only need one bit.
  */

private:
  friend class APT_SC; // access to scInterface()
  // friend class APT_ProcessManager;  // also for access to scInterface()
  friend class APT_StepSC;
  friend class APT_StepRep;
  friend class APT_Operator;
  friend class APT_Operator_UT; // for testing
  friend class APT_IR;

  // prohibit copying
  APT_Step(const APT_Step&);
  APT_Step& operator= (const APT_Step&);
```

```
  const APT_StepSC* scInterface() const;
  APT_StepSC* scInterface();
  /*
     effect    Returns a pointer to an object providing supplemental
               interface to this step.
     note      The caller must not delete the returned pointer.
  */

  APT_StepRep* rep_;
};

#endif // APT_STEP_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_SUBCURSOR_H
#define APT_SUBCURSOR_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_ERRLOG_H
#include <apt_util/errlog.h>
#endif

#ifndef APT_FIELDSEL_H
#include <apt_framework/fieldsel.h>
#endif

#ifndef APT_TAGACCESSOR_H
#include <apt_framework/tagaccessor.h> // for APT_ScopeAccessorTarget
#endif

class APT_InputAccessorInterface;
class APT_OutputAccessorInterface;
class APT_FieldSelector;

class APT_InterfaceRep;

class APT_SubCursorBase
{
public:
  virtual ~APT_SubCursorBase();

  bool isSetup() const { return target_ != 0; }

  bool isUsable() const;
  /*
     effect    Tells if this sub-cursor is currently usable.
               False if:
               - isSetup() is false, or
               - this is an input sub-cursor and
                 APT_InputCursor::getRecord() has never been called
                 for the dataset to which this sub-cursor is bound, or
               - this is an input sub-cursor and it is bound to a subrec
                 within a tagged aggregate that is not "active".
```

```
  */
  bool operator! () const { return !isUsable(); }

  bool isFixedLengthVector() const;
  /*
     effect    Tells whether this sub-cursor is bound to a subrec
               that a fixed-length vector of instances.
     requires  isSetup() is true.
  */

  APT_UInt32 vectorLength() const;
  /*
     effect    Gives the vector length of the subrec to which this
               accessor is bound.
               Returns 1 if the field is not a vector.
     requires  isUsable() or isFixedLengthVector() is true
  */

  APT_UInt32 position() const;
  /*
     effect    Returns the current position of this sub-cursor.
     requires  isUsable() is true.
               vectorLength() > 0
  */

  APT_FieldSelector concreteField() const;
  /*
     effect    Returns the name of the data set subrec to which
               this sub-cursor is bound.
     requires  isSetup() is true.
  */

  friend APT_Archive& operator|| (APT_Archive& ar, APT_SubCursorBase&)
  { return ar; }

protected:
  friend class APT_InterfaceRep;

  APT_SubCursorBase();

  APT_ScopeAccessorTarget* target_;

private:
  // prohibit copy/assign
  APT_SubCursorBase(const APT_SubCursorBase&);
  APT_SubCursorBase& operator= (const APT_SubCursorBase&);
```

```
};

class APT_InputSubCursor : public APT_SubCursorBase
{
public:
  APT_InputSubCursor();
  /*
     effect    Constructs an input subcursor, which must be bound to
               a subrec component of an input cursor before it can be
               used.
  */
  APT_InputSubCursor(const APT_FieldSelector& component,
                     APT_InputAccessorInterface* cur);
  /*
     effect    Initializes this subcursor object to the indicated
               subrec component accessible using the given cursor
               object.
     requires  cur non-null
               cur->isSetup() is true.
               component must identify a subrec of the cursor's
               schema().
               component must not be subscripted.
               The indicated component must not have been setup with
               a subcursor already.
  */

  bool next();
  bool prev();
  /* moves in the indicated direction; returns false if no movement
     is possible.
     requires  isUsable() is true.
               vectorLength() > 0
     fatal: unhandled input conversion failure
  */

  void setPosition(APT_UInt32 pos);
  /* requires: 0 <= pos < vectorLength()
     requires  isUsable() is true.
     fatal: unhandled input conversion failure
  */

private:
  APT_ErrorLog log_;
};

class APT_OutputSubCursor : public APT_SubCursorBase
```

```
{
public:
  APT_OutputSubCursor();
  /*
     effect    Constructs an output subcursor, which must be bound to
               a subrec component of an output cursor before it can be
               used.
  */
  APT_OutputSubCursor(const APT_FieldSelector& component,
                      APT_OutputAccessorInterface* cur);
  /*
     effect    Initializes this subcursor object to the indicated
               subrec component accessible using the given cursor
               object.
     requires  cur non-null
               cur->isSetup() is true.
               component must identify a subrec of the cursor's
               schema().
               component must not be subscripted.
               The indicated component must not have been setup with
               a subcursor already.
  */

  bool next();
  bool prev();
  // requires: isUsable() true, vectorLength() > 0

  void setPosition(APT_UInt32 pos);
  /* requires: 0 <= pos < vectorLength()
               isUsable() true
  */

  void setVectorLength(APT_UInt32);
  /*
     effect    Sets the length of the subrec vector to which this
               sub-cursor is bound.
               If the length is reduced, subrec elements at the top of
               the vector are discarded.  If the length is increased,
               the new subrec elements at the top of the vector are
               cleared/nulled.
     requires  0 <= len <= 0x7fffffff
               isVector() is true.
               isFixedLengthVector() is false.
               isUsable() is true.
  */
};
```

```
#endif // APT_SUBCURSOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TAGACCESSOR_H
#define APT_TAGACCESSOR_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_FIELDSEL_H
#include <apt_framework/fieldsel.h>
#endif

class APT_ErrorLog;
class APT_SubCursorBase;
class APT_InputSubCursor;
class APT_OutputSubCursor;
class APT_InputTagAccessor;
class APT_OutputTagAccessor;
class APT_InterfaceRep;

class APT_ScopeAccessorTarget
/* Synthetic base class that we mix into the APT_Interface internals, so
   that we can hide most of the APT_Interface implementation and yet have
   these crucial accessor-related functions be inline.  For the tag
   accessor and sub-cursor classes.

   Torrent note: APT_InterfaceRep::Scope is the only class that may have
   APT_ScopeAccessorTarget as a base class.
*/
{
protected:
  friend class APT_InputTagAccessor;
  friend class APT_OutputTagAccessor;
  friend class APT_SubCursorBase;
  friend class APT_InputSubCursor;
  friend class APT_OutputSubCursor;
  friend class APT_InterfaceRep;

  APT_ScopeAccessorTarget();
  ~APT_ScopeAccessorTarget();

  APT_UInt32 vecMode_f;         /* 0: var-length; positive: fixed-length */
  const bool* presentp_;        /* points to flag that will be false if
                                   this scope is within a scope vector
                                   whose number of occurrences is
                                   zero; the flag updated to point into this
                                   component's concrete scope. */
  const bool* tagActivep_;      /* points to flag updated if this
                                   scope's concrete counterpart is an
```

```
                                    arm of a tagged; always true otherwise */
  APT_UInt32* activeLenp_;               /* for var-len subrec vector, elements of
                                    vector actually active for this record;
                                    if non-null, always points to concrete
                                    control scope's scopeVecLen_ */
public:
  int numCases_f;
  int tagVal_;

protected:
  APT_FieldSelector concreteField() const;
  void setTag_(int tagVal);

  APT_UInt32 curPosition_() const;
  void setVectorLength_(APT_UInt32);
  void positionInputSubCursor_(APT_UInt32 pos, APT_ErrorLog&);
  void positionOutputSubCursor_(APT_UInt32 pos, bool forClearing);

public:
  APT_UInt32 vecMode() const { return vecMode_f; }
  int numCases() const { return numCases_f; }

  bool isUsable() const { return *presentp_ && *tagActivep_; }

private:
  // prohibit copy/assign
  APT_ScopeAccessorTarget(const APT_ScopeAccessorTarget&);
  APT_ScopeAccessorTarget& operator= (const APT_ScopeAccessorTarget&);
};


class APT_InputTagAccessor
/* An input tag accessor is used to read a tagged aggregate's tag
   value.
*/
{
public:
  APT_InputTagAccessor();
  ~APT_InputTagAccessor();

  bool isSetup() const { return target_ != 0; }
  /*
     effect    Tells if this tag accessor has been bound to an interface
               component.
  */

  bool isUsable() const;
  /*
     effect    Tells if this tag accessor is currently usable to access
               a tag value.
               False if:
               - isSetup() is false, or
```

```
                - this is an input tag accessor and
                  APT_InputCursor::getRecord() has never been called
                  for the dataset to which this accessor is bound
                - this is an input tag accessor and it is bound to a field
                  within a tagged aggregate that is not "active".
  */
  bool operator! () const { return !isUsable(); }

  bool isFixedTag() const;
  /*
     effect    Tells whether this field accessor is bound to a tagged
               union whose tag is constant.
     requires  isSetup() is true.
     DEPRECATED-- now always returns false
  */

  int numTags() const;
  /*
     effect    Tells how many tag values are available for the tagged
               aggregate to which this accessor is bound.
     requires  isSetup() is true.
  */

  int tag() const;
  /*
     effect    Returns the tag value of the tagged union to which this
               tag accessor is bound.
     requires  isFixedTag() is true or isUsable() is true.
  */

  APT_FieldSelector concreteField() const;
  /*
     effect    Returns the name of the dataset tagged aggregate to
               which this tag accessor is bound.
     requires  isSetup() is true.
  */

protected:
  friend class APT_InterfaceRep;

  APT_ScopeAccessorTarget* target_;

private:
  // prohibit copying
  APT_InputTagAccessor(const APT_InputTagAccessor&);
  APT_InputTagAccessor& operator= (const APT_InputTagAccessor&);
};

class APT_OutputTagAccessor : public APT_InputTagAccessor
/* adds the ability to set the tag value */
{
public:
```

```
  // default ctor, dtor OK; copy/assign prohibited in base class

  void setTag(int tag);
  /*
     effect    Sets the tag value of the tagged union to which this
               tag accessor is bound.
     note      An output tagged union's tag must be set before
               writing any fields of the tagged union.
     requires  0 <= tag < numTags()
               isUsable() is true.
  */
};

class APT_TagAccessor : public APT_OutputTagAccessor
// back-compatibility
{
};

#endif // APT_TAGACCESSOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_CONVERSIONS_DEFAULT_H
#define APT_CONVERSIONS_DEFAULT_H

#ifndef APT_CONVERSION_H
#include <apt_framework/type/conversion.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

class APT_String;
class APT_RawField;

/* declare an extern symbol that can be referenced from elsewhere, in
   order to cause the 'conversions_default.C' compilation unit to be
   pulled in by the linker */
extern int sAPT_DefaultConversions;

/* Declarations for default conversions among most "basic" (v1.1)
   Orchestrate field types. */

// naming convention: APT_DefaultConversion_DestType_SrcType

#define APT_DECLARE_DEFAULT_CONVERSION(DT,ST)                              \
class APT_DefaultConversion_ ## DT ## _ ## ST : public APT_FieldConversion \
{                                                                          \
  APT_DECLARE_RTTI(APT_DefaultConversion_ ## DT ## _ ## ST);               \
  APT_DECLARE_PERSISTENT(APT_DefaultConversion_ ## DT ## _ ## ST);         \
                                                                           \
public:                                                                    \
  APT_DefaultConversion_ ## DT ## _ ## ST();                               \
  /* default copy/assign OK */                                             \
                                                                           \
  virtual APT_Status convert(const void* STval, void* DTval, void*) const;\
                                                                           \
  static int registerDescriptor(); /* initialization support */           \
  /* first call registers this type; subsequent calls are nops */         \
protected:                                                                 \
  virtual APT_FieldConversion* clone() const;                              \
};                                                                         \
static int APT_sRegisterDefaultConversion_ ## DT ## _ ## ST =              \
  APT_DefaultConversion_ ## DT ## _ ## ST::registerDescriptor()
/* The above causes every compilation unit that includes this header
   file to call the registerDesriptor() function for the conversion
```

```
   classes.  In this manner, we ensure that anyone who wants to use
   these conversion classes will always find that they have been
   registered. */

// as above, with conversion warning
#define APT_DECLARE_DEFAULT_CONVERSION_WARN(DT,ST)                        \
class APT_DefaultConversion_ ## DT ## _ ## ST : public APT_FieldConversion \
{                                                                         \
  APT_DECLARE_RTTI(APT_DefaultConversion_ ## DT ## _ ## ST);              \
  APT_DECLARE_PERSISTENT(APT_DefaultConversion_ ## DT ## _ ## ST);        \
                                                                          \
public:                                                                   \
  APT_DefaultConversion_ ## DT ## _ ## ST();                              \
  /* default copy/assign OK */                                            \
                                                                          \
  virtual Validation validate(const APT_FieldTypeDescriptor* src,         \
                              const APT_FieldTypeDescriptor* dest,        \
                              APT_String* reason);                 \
  virtual APT_Status convert(const void* STval, void* DTval, void*) const;\
                                                                          \
  static int registerDescriptor(); /* initialization support */          \
  /* first call registers this type; subsequent calls are nops */        \
protected:                                                                \
  virtual APT_FieldConversion* clone() const;                             \
};                                                                        \
static int APT_sRegisterDefaultConversion_ ## DT ## _ ## ST =             \
  APT_DefaultConversion_ ## DT ## _ ## ST::registerDescriptor()


//APT_DECLARE_DEFAULT_CONVERSION(UInt8, UInt8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(UInt16, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(Int16, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(UInt32, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(Int32, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(UInt64, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(Int64, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(SFloat, UInt8);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, UInt8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, UInt8);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, Int8);
//APT_DECLARE_DEFAULT_CONVERSION(Int8, Int8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, Int8);
APT_DECLARE_DEFAULT_CONVERSION(Int16, Int8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, Int8);
APT_DECLARE_DEFAULT_CONVERSION(Int32, Int8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, Int8);
APT_DECLARE_DEFAULT_CONVERSION(Int64, Int8);
```

```
APT_DECLARE_DEFAULT_CONVERSION(SFloat, Int8);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, Int8);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, Int8);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, UInt16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, UInt16);
//APT_DECLARE_DEFAULT_CONVERSION(UInt16, UInt16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(UInt32, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(Int32, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(UInt64, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(Int64, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(SFloat, UInt16);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, UInt16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, UInt16);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, Int16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, Int16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, Int16);
//APT_DECLARE_DEFAULT_CONVERSION(Int16, Int16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, Int16);
APT_DECLARE_DEFAULT_CONVERSION(Int32, Int16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, Int16);
APT_DECLARE_DEFAULT_CONVERSION(Int64, Int16);
APT_DECLARE_DEFAULT_CONVERSION(SFloat, Int16);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, Int16);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, Int16);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, UInt32);
//APT_DECLARE_DEFAULT_CONVERSION(UInt32, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, UInt32);
APT_DECLARE_DEFAULT_CONVERSION(UInt64, UInt32);
APT_DECLARE_DEFAULT_CONVERSION(Int64, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, UInt32);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, UInt32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, UInt32);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, Int32);
//APT_DECLARE_DEFAULT_CONVERSION(Int32, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, Int32);
APT_DECLARE_DEFAULT_CONVERSION(Int64, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, Int32);
```

```
APT_DECLARE_DEFAULT_CONVERSION(DFloat, Int32);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, Int32);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, UInt64);
//APT_DECLARE_DEFAULT_CONVERSION(UInt64, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int64, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(DFloat, UInt64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, UInt64);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, Int64);
//APT_DECLARE_DEFAULT_CONVERSION(Int64, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(DFloat, Int64);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, Int64);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int64, SFloat);
//APT_DECLARE_DEFAULT_CONVERSION(SFloat, SFloat);
APT_DECLARE_DEFAULT_CONVERSION(DFloat, SFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, SFloat);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int64, DFloat);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, DFloat);
//APT_DECLARE_DEFAULT_CONVERSION(DFloat, DFloat);
```

```
APT_DECLARE_DEFAULT_CONVERSION_WARN(String, DFloat);

APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt8, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int8, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt16, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int16, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt32, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int32, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(UInt64, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(Int64, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(SFloat, String);
APT_DECLARE_DEFAULT_CONVERSION_WARN(DFloat, String);
//APT_DECLARE_DEFAULT_CONVERSION(String, String);


/* Declarations for default conversions among the "basic" (v1.1)
   Orchestrate field types, but involving some special-case behavior. */

class APT_DefaultConversion_String_String : public APT_FieldConversion
{
  APT_DECLARE_RTTI(APT_DefaultConversion_String_String);
  APT_DECLARE_PERSISTENT(APT_DefaultConversion_String_String);

public:
  APT_DefaultConversion_String_String();
  /* default copy/assign OK */

  virtual Validation validate(const APT_FieldTypeDescriptor* src,
                              const APT_FieldTypeDescriptor* dest,
                              APT_String* reason);

  virtual APT_Status convert(const void* STval, void* DTval, void*) const;

  static int registerDescriptor(); /* initialization support */
  /* first call registers this type; subsequent calls are nops */

protected:
  virtual APT_FieldConversion* clone() const;
};
static int APT_sRegisterDefaultConversion_String_String =
  APT_DefaultConversion_String_String::registerDescriptor();


class APT_DefaultConversion_Raw_Raw : public APT_FieldConversion
{
  APT_DECLARE_RTTI(APT_DefaultConversion_Raw_Raw);
  APT_DECLARE_PERSISTENT(APT_DefaultConversion_Raw_Raw);

public:
```

```
  APT_DefaultConversion_Raw_Raw();
  /* default copy/assign OK */

  virtual Validation validate(const APT_FieldTypeDescriptor* src,
                              const APT_FieldTypeDescriptor* dest,
                              APT_String* reason);

  virtual APT_Status convert(const void* STval, void* DTval, void*) const;

  static int registerDescriptor(); /* initialization support */
  /* first call registers this type; subsequent calls are nops */

protected:
  virtual APT_FieldConversion* clone() const;
};
static int APT_sRegisterDefaultConversion_Raw_Raw =
  APT_DefaultConversion_Raw_Raw::registerDescriptor();

#endif // APT_CONVERSIONS_DEFAULT_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_FLOAT_H
#define APT_FLOAT_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_FUNCTION_H
#include <apt_framework/type/function.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

/* Declarations for the built-in floating point field types:
   sfloat, dfloat
*/

#define APT_DECLARE_FLOAT_DESCRIPTOR(T)                                  \
class APT_ ## T ## Descriptor : public APT_FieldTypeDescriptor           \
{                                                                        \
  APT_DECLARE_RTTI(APT_ ## T ## Descriptor);                             \
  APT_DECLARE_PERSISTENT(APT_ ## T ## Descriptor);                       \
                                                                         \
public:                                                                  \
  APT_ ## T ## Descriptor();                                             \
  /* default copy/assign OK */                                           \
  virtual void* allocValueType(APT_UInt32 veclen) const;                 \
  virtual void freeValueType(void* Tvec, APT_UInt32 veclen) const;       \
                                                                         \
  virtual void copyValueType(const void* Tsrc, void* Tdest) const;       \
  virtual void copyValueType_vec(const void* Tsrc, void* Tdest,          \
                                 APT_UInt32 veclen) const;               \
                                                                         \
  virtual void clearValueType(void* Tdest) const;                        \
  virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;\
                                                                         \
  virtual void setInBandNull(void* Tdest) const;                         \
                                                                         \
  virtual bool isBuiltInType() const;   /* true for floats */            \
                                                                         \
  static int registerDescriptor(); /* initialization support */         \
  /* first call registers this type; subsequent calls are nops */       \
                                                                         \
  virtual APT_FieldProtocol* protocol() const;                           \
```

```
                                                                        \
protected:                                                              \
  virtual APT_FieldTypeDescriptor* clone() const;                       \
};                                                                      \
static int APT_sRegister ## T =                                         \
  APT_ ## T ## Descriptor::registerDescriptor()

/* The above causes every compilation unit that includes this header
   file to call the registerDesriptor() function for the descriptor
   classes.  In this manner, we ensure that anyone who wants to use
   these descriptor classes will always find that they have been
   registered. */

APT_DECLARE_FLOAT_DESCRIPTOR(SFloat);
APT_DECLARE_FLOAT_DESCRIPTOR(DFloat);


/* like APT_DECLARE_ACCESSORS defined in accessorbase.h, except
   this form doesn't declare -> operators */
#define APT_DECLARE_FLT_ACCESSORS(VT, ST, AT)                           \
class APT_InputAccessorTo ## AT : public APT_InputAccessorBase          \
{ APT_DECLARE_RTTI(APT_InputAccessorTo ## AT); /* DEPRECATED */         \
public:                                                                 \
  APT_InputAccessorTo ## AT();                                          \
  APT_InputAccessorTo ## AT(const APT_FieldSelector& component,         \
                            APT_InputAccessorInterface* cur);           \
  const VT& value() const { return *base(0); }                          \
  const VT& operator* () const { return value(); }                      \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }            \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }      \
private:                                                                \
  const VT* base(APT_UInt32 i) const                                    \
    { return (const VT*) APT_InputAccessorBase::valueAt(i); }           \
};                                                                      \
class APT_OutputAccessorTo ## AT : public APT_OutputAccessorBase        \
{ APT_DECLARE_RTTI(APT_OutputAccessorTo ## AT); /* DEPRECATED */        \
public:                                                                 \
  APT_OutputAccessorTo ## AT();                                         \
  APT_OutputAccessorTo ## AT(const APT_FieldSelector& component,        \
                            APT_OutputAccessorInterface* cur);          \
  const VT& value() const { return *base(0); }                          \
  VT& writableValue() { return *base(0); }                              \
  const VT& operator* () const { return value(); }                      \
  VT& operator* () { return *base(0); }                                 \
  void setValue(const VT& val) { *base(0) = val; }                      \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }            \
  VT& writableValueAt(APT_UInt32 i) { return *base(i); }                \
  void setValueAt(APT_UInt32 i, const VT& val) { *base(i) = val; }      \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }      \
  VT& operator[] (APT_UInt32 i) { return *base(i); }                    \
private:                                                                \
  const VT* base(APT_UInt32 i) const                                    \
    { return (const VT*) APT_AccessorBase::base(i); }                   \
  VT* base(APT_UInt32 i) { return (VT*) APT_OutputAccessorBase::valueAt(i); }\
```

```
}

APT_DECLARE_FLT_ACCESSORS(APT_SFloat, sfloat, SFloat);
APT_DECLARE_FLT_ACCESSORS(APT_DFloat, dfloat, DFloat);

// for back-compatibility

#ifndef APT_ACCESSOR_TEMPLATES_DEF
#define APT_ACCESSOR_TEMPLATES_DEF
template<class T> class APT_InputAccessor {};
template<class T> class APT_OutputAccessor {};
#endif

class APT_InputAccessor<APT_SFloat> : public APT_InputAccessorToSFloat
{
public:
  APT_InputAccessor() : APT_InputAccessorToSFloat() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToSFloat(comp, cur) {}
};
class APT_OutputAccessor<APT_SFloat> : public APT_OutputAccessorToSFloat
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToSFloat() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToSFloat(comp, cur) {}
};

class APT_InputAccessor<APT_DFloat> : public APT_InputAccessorToDFloat
{
public:
  APT_InputAccessor() : APT_InputAccessorToDFloat() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToDFloat(comp, cur) {}
};
class APT_OutputAccessor<APT_DFloat> : public APT_OutputAccessorToDFloat
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToDFloat() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToDFloat(comp, cur) {}
};


/**** declarations for basic generic functions on floating point values ****/
```

```
#define APT_DECLARE_FLOAT_FUNCTIONS(T)                                  \
class APT_ ## T ## GFEquality : public APT_GFEquality                   \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFEquality);                            \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFEquality);                      \
public:                                                                 \
  APT_ ## T ## GFEquality();                                            \
  /* default copy/assign OK */                                          \
  virtual bool isEqual(const void* Tleft, const void* Tright) const;    \
  virtual APT_UInt32 hash(const void* Tval) const;                      \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */           \
  /* first call registers this func; subsequent calls are nops */       \
};                                                                      \
static int APT_sRegister ## T ## GFEquality=                            \
  APT_ ## T ## GFEquality::registerFunction();                          \
                                                                        \
class APT_ ## T ## GFComparison : public APT_GFComparison               \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFComparison);                          \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFComparison);                    \
public:                                                                 \
  APT_ ## T ## GFComparison();                                          \
  /* default copy/assign OK */                                          \
  virtual CompareResult compare(const void* Tleft, const void* Tright) const; \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */           \
  /* first call registers this func; subsequent calls are nops */       \
};                                                                      \
static int APT_sRegister ## T ## GFComparison=                          \
  APT_ ## T ## GFComparison::registerFunction();                        \
                                                                        \
class APT_ ## T ## GFPrint : public APT_GFPrint                         \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFPrint);                               \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFPrint);                         \
public:                                                                 \
  APT_ ## T ## GFPrint();                                               \
  /* default copy/assign OK */                                          \
  virtual void print(const void* Tval, ostream&) const;                 \
  virtual APT_Status scan(const APT_String& literal, void* Tval);       \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */           \
  /* first call registers this func; subsequent calls are nops */       \
};                                                                      \
static int APT_sRegister ## T ## GFPrint=                               \
  APT_ ## T ## GFPrint::registerFunction()

APT_DECLARE_FLOAT_FUNCTIONS(SFloat);
APT_DECLARE_FLOAT_FUNCTIONS(DFloat);

#if ((defined (PROFILE)) || ((defined(__GNUC__) && ! defined(__LINUX__))))
class APT_GFIX_FloatRegistration
{
```

```
public:
  static int nopForRegistration()
  { extern int APT_GFIX_FloatMagicCookie; return APT_GFIX_FloatMagicCookie; };
};

static int APT_GFIX_registerFloat=APT_GFIX_FloatRegistration::nopForRegistration();
#endif

#endif // APT_FLOAT_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_INTEGER_H
#define APT_INTEGER_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_FUNCTION_H
#include <apt_framework/type/function.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif


/* Declarations for the built-in integer field types:
   int8, uint8, int16, uint16, int32, uint32, int64, uint64
*/

#define APT_DECLARE_INTEGER_DESCRIPTOR(T)                                \
class APT_ ## T ## Descriptor : public APT_FieldTypeDescriptor           \
{                                                                        \
  APT_DECLARE_RTTI(APT_ ## T ## Descriptor);                             \
  APT_DECLARE_PERSISTENT(APT_ ## T ## Descriptor);                       \
                                                                         \
public:                                                                  \
  APT_ ## T ## Descriptor();                                             \
  /* default copy/assign OK */                                           \
  virtual void* allocValueType(APT_UInt32 veclen) const;                 \
  virtual void freeValueType(void* Tvec, APT_UInt32 veclen) const;       \
                                                                         \
  virtual void copyValueType(const void* Tsrc, void* Tdest) const;       \
  virtual void copyValueType_vec(const void* Tsrc, void* Tdest,          \
                                 APT_UInt32 veclen) const;               \
                                                                         \
  virtual void clearValueType(void* Tdest) const;                        \
  virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;\
                                                                         \
  virtual void setInBandNull(void* Tdest) const;                         \
                                                                         \
  virtual bool isBuiltInType() const;   /* true for integers */          \
                                                                         \
```

```
  static int registerDescriptor(); /* initialization support */         \
  /* first call registers this type; subsequent calls are nops */       \
                                                                        \
  virtual APT_FieldProtocol* protocol() const;                          \
                                                                        \
protected:                                                              \
  virtual APT_FieldTypeDescriptor* clone() const;                       \
};                                                                      \
static int APT_sRegister ## T =                                         \
  APT_ ## T ## Descriptor::registerDescriptor()

/* The above causes every compilation unit that includes this header
   file to call the registerDesriptor() function for the descriptor
   classes.  In this manner, we ensure that anyone who wants to use
   these descriptor classes will always find that they have been
   registered. */

APT_DECLARE_INTEGER_DESCRIPTOR(Int8);
APT_DECLARE_INTEGER_DESCRIPTOR(UInt8);
APT_DECLARE_INTEGER_DESCRIPTOR(Int16);
APT_DECLARE_INTEGER_DESCRIPTOR(UInt16);
APT_DECLARE_INTEGER_DESCRIPTOR(Int32);
APT_DECLARE_INTEGER_DESCRIPTOR(UInt32);
APT_DECLARE_INTEGER_DESCRIPTOR(Int64);
APT_DECLARE_INTEGER_DESCRIPTOR(UInt64);



/* like APT_DECLARE_ACCESSORS defined in accessorbase.h, except
   this form doesn't declare -> operators */
#define APT_DECLARE_INT_ACCESSORS(VT, ST, AT)                           \
class APT_InputAccessorTo ## AT : public APT_InputAccessorBase          \
{ APT_DECLARE_RTTI(APT_InputAccessorTo ## AT); /* DEPRECATED */         \
public:                                                                 \
  APT_InputAccessorTo ## AT();                                          \
  APT_InputAccessorTo ## AT(const APT_FieldSelector& component,         \
                            APT_InputAccessorInterface* cur);           \
  const VT& value() const { return *base(0); }                          \
  const VT& operator* () const { return value(); }                      \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }            \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }      \
private:                                                                \
  const VT* base(APT_UInt32 i) const                                    \
    { return (const VT*) APT_InputAccessorBase::valueAt(i); }           \
};                                                                      \
class APT_OutputAccessorTo ## AT : public APT_OutputAccessorBase        \
{ APT_DECLARE_RTTI(APT_OutputAccessorTo ## AT); /* DEPRECATED */        \
public:                                                                 \
  APT_OutputAccessorTo ## AT();                                         \
  APT_OutputAccessorTo ## AT(const APT_FieldSelector& component,        \
                             APT_OutputAccessorInterface* cur);         \
```

```
  const VT& value() const { return *base(0); }                              \
  VT& writableValue() { return *base(0); }                                  \
  const VT& operator* () const { return value(); }                          \
  VT& operator* () { return *base(0); }                                     \
  void setValue(const VT& val) { *base(0) = val; }                          \
  const VT& valueAt(APT_UInt32 i) const { return *base(i); }                \
  VT& writableValueAt(APT_UInt32 i) { return *base(i); }                    \
  void setValueAt(APT_UInt32 i, const VT& val) { *base(i) = val; }          \
  const VT& operator[] (APT_UInt32 i) const { return valueAt(i); }          \
  VT& operator[] (APT_UInt32 i) { return *base(i); }                        \
private:                                                                    \
  const VT* base(APT_UInt32 i) const                                        \
    { return (const VT*) APT_AccessorBase::base(i); }                       \
  VT* base(APT_UInt32 i) { return (VT*) APT_OutputAccessorBase::valueAt(i); }\
}

APT_DECLARE_INT_ACCESSORS(APT_Int8, int8, Int8);
APT_DECLARE_INT_ACCESSORS(APT_UInt8, uint8, UInt8);
APT_DECLARE_INT_ACCESSORS(APT_Int16, int16, Int16);
APT_DECLARE_INT_ACCESSORS(APT_UInt16, uint16, UInt16);
APT_DECLARE_INT_ACCESSORS(APT_Int32, int32, Int32);
APT_DECLARE_INT_ACCESSORS(APT_UInt32, uint32, UInt32);
APT_DECLARE_INT_ACCESSORS(APT_Int64, int64, Int64);
APT_DECLARE_INT_ACCESSORS(APT_UInt64, uint64, UInt64);


// for back-compatibility

#ifndef APT_ACCESSOR_TEMPLATES_DEF
#define APT_ACCESSOR_TEMPLATES_DEF
template<class T> class APT_InputAccessor {};
template<class T> class APT_OutputAccessor {};
#endif

class APT_InputAccessor<APT_Int8> : public APT_InputAccessorToInt8
{
public:
  APT_InputAccessor() : APT_InputAccessorToInt8() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToInt8(comp, cur) {}
};
class APT_OutputAccessor<APT_Int8> : public APT_OutputAccessorToInt8
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToInt8() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToInt8(comp, cur) {}
```

```
};

class APT_InputAccessor<APT_UInt8> : public APT_InputAccessorToUInt8
{
public:
  APT_InputAccessor() : APT_InputAccessorToUInt8() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToUInt8(comp, cur) {}
};
class APT_OutputAccessor<APT_UInt8> : public APT_OutputAccessorToUInt8
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToUInt8() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToUInt8(comp, cur) {}
};

class APT_InputAccessor<APT_Int16> : public APT_InputAccessorToInt16
{
public:
  APT_InputAccessor() : APT_InputAccessorToInt16() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToInt16(comp, cur) {}
};
class APT_OutputAccessor<APT_Int16> : public APT_OutputAccessorToInt16
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToInt16() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToInt16(comp, cur) {}
};

class APT_InputAccessor<APT_UInt16> : public APT_InputAccessorToUInt16
{
public:
  APT_InputAccessor() : APT_InputAccessorToUInt16() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToUInt16(comp, cur) {}
};
class APT_OutputAccessor<APT_UInt16> : public APT_OutputAccessorToUInt16
{
```

```
public:
  APT_OutputAccessor() : APT_OutputAccessorToUInt16() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToUInt16(comp, cur) {}
};

class APT_InputAccessor<APT_Int32> : public APT_InputAccessorToInt32
{
public:
  APT_InputAccessor() : APT_InputAccessorToInt32() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToInt32(comp, cur) {}
};
class APT_OutputAccessor<APT_Int32> : public APT_OutputAccessorToInt32
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToInt32() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToInt32(comp, cur) {}
};

class APT_InputAccessor<APT_UInt32> : public APT_InputAccessorToUInt32
{
public:
  APT_InputAccessor() : APT_InputAccessorToUInt32() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToUInt32(comp, cur) {}
};
class APT_OutputAccessor<APT_UInt32> : public APT_OutputAccessorToUInt32
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToUInt32() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToUInt32(comp, cur) {}
};


class APT_InputAccessor<APT_Int64> : public APT_InputAccessorToInt64
{
public:
  APT_InputAccessor() : APT_InputAccessorToInt64() {}
```

```
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToInt64(comp, cur) {}
};
class APT_OutputAccessor<APT_Int64> : public APT_OutputAccessorToInt64
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToInt64() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToInt64(comp, cur) {}
};

class APT_InputAccessor<APT_UInt64> : public APT_InputAccessorToUInt64
{
public:
  APT_InputAccessor() : APT_InputAccessorToUInt64() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToUInt64(comp, cur) {}
};
class APT_OutputAccessor<APT_UInt64> : public APT_OutputAccessorToUInt64
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToUInt64() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToUInt64(comp, cur) {}
};



/**** declarations for basic generic functions on integers ****/

#define APT_DECLARE_INTEGER_FUNCTIONS(T)                                \
class APT_ ## T ## GFEquality : public APT_GFEquality                   \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFEquality);                            \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFEquality);                      \
public:                                                                 \
  APT_ ## T ## GFEquality();                                            \
  /* default copy/assign OK */                                          \
  virtual bool isEqual(const void* Tleft, const void* Tright) const;    \
  virtual APT_UInt32 hash(const void* Tval) const;                      \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */          \
  /* first call registers this func; subsequent calls are nops */       \
```

```
};                                                                      \
static int APT_sRegister ## T ## GFEquality=                            \
  APT_ ## T ## GFEquality::registerFunction();                          \
                                                                        \
class APT_ ## T ## GFComparison : public APT_GFComparison               \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFComparison);                          \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFComparison);                    \
public:                                                                 \
  APT_ ## T ## GFComparison();                                          \
  /* default copy/assign OK */                                          \
  virtual CompareResult compare(const void* Tleft, const void* Tright) const; \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */          \
  /* first call registers this func; subsequent calls are nops */       \
};                                                                      \
static int APT_sRegister ## T ## GFComparison=                          \
  APT_ ## T ## GFComparison::registerFunction();                        \
                                                                        \
class APT_ ## T ## GFPrint : public APT_GFPrint                         \
{                                                                       \
  APT_DECLARE_RTTI(APT_ ## T ## GFPrint);                               \
  APT_DECLARE_PERSISTENT(APT_ ## T ## GFPrint);                         \
public:                                                                 \
  APT_ ## T ## GFPrint();                                               \
  /* default copy/assign OK */                                          \
  virtual void print(const void* Tval, ostream&) const;                 \
  virtual APT_Status scan(const APT_String& literal, void* Tval);       \
  virtual APT_GenericFunction* clone() const;                           \
  static int registerFunction(); /* initialization support */          \
  /* first call registers this func; subsequent calls are nops */       \
};                                                                      \
static int APT_sRegister ## T ## GFPrint=                               \
  APT_ ## T ## GFPrint::registerFunction()

APT_DECLARE_INTEGER_FUNCTIONS(Int8);
APT_DECLARE_INTEGER_FUNCTIONS(UInt8);
APT_DECLARE_INTEGER_FUNCTIONS(Int16);
APT_DECLARE_INTEGER_FUNCTIONS(UInt16);
APT_DECLARE_INTEGER_FUNCTIONS(Int32);
APT_DECLARE_INTEGER_FUNCTIONS(UInt32);
APT_DECLARE_INTEGER_FUNCTIONS(Int64);
APT_DECLARE_INTEGER_FUNCTIONS(UInt64);


#if ((defined (PROFILE)) || ((defined(__GNUC__) && ! defined(__LINUX__))))
class APT_GFIX_IntRegistration
{
public:
  static int nopForRegistration()
  { extern int APT_GFIX_IntMagicCookie; return APT_GFIX_IntMagicCookie; };
```

```
};

static int APT_GFIX_registerInt=APT_GFIX_IntRegistration::nopForRegistration();
#endif

#endif // APT_INTEGER_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_RAW_H
#define APT_RAW_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_FUNCTION_H
#include <apt_framework/type/function.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_RAWFIELD_H
#include <apt_framework/rawfield.h>
#endif

#ifndef APT_SCHEMA_H
#include <apt_framework/schema.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_ErrorLog;

class APT_RawFieldDescriptor : public APT_FieldTypeDescriptor
{
public:
  APT_RawFieldDescriptor();
  ~APT_RawFieldDescriptor();

  APT_RawFieldDescriptor(const APT_RawFieldDescriptor&);
  APT_RawFieldDescriptor& operator= (const APT_RawFieldDescriptor&);

  virtual APT_String accessorTypeNameFragment() const;

  bool isFixedLength() const { return isFixedLength_; }
  APT_UInt32 fixedLength() const;       // requires: isFixedLength() true
```

```
bool isBoundedLength() const { return isBoundedLength_; }
APT_UInt32 boundedLength() const;     // requires: isBoundedLength() true

void setVariableLength();
void setFixedLength(APT_UInt32);
void setBoundedLength(APT_UInt32);
// must be non-zero; hasSchema() must be false

int alignment() const { return align_; }

void setAlignment(int);       // must be power of 2
// hasSchema() must be false

bool hasSchema() const { return schema_ ? true : false; }
/*
   effect    Indicates whether this raw is parameterized by an
             import schema (meaning this raw represents an intact
             record).
*/
APT_Schema schema() const;
/*
   effect    Returns the import schema for this raw.
   note      The returned schema is guaranteed to be a valid import
             schema.
   requires  hasSchema() must be true.
*/
void setSchema(const APT_Schema& sch, APT_ErrorLog&);
/*
   effect    Sets the import schema for this raw (meaning this raw
             represents an intact record).
             hasSchema() becomes true.
   note      Any errors (such as invalid import schema, or nested
             intact record) are reported via the supplied error log
             object.
             Regardless of whether the indicated import schema is
             fixed-length, this raw will be variable-length.
   requires  sch.isIntact() must be false.
             sch.isConcreteSchema() must be true.
             sch's record type must not be "implicit" or
             "streamed".
*/
APT_Schema intfSchema() const;
/*
   effect    Returns the interface schema corresponding to the
             schema() import schema.
   requires  hasSchema() must be true.
*/

virtual void* allocValueType(APT_UInt32 veclen) const;
virtual void freeValueType(void* Tvec, APT_UInt32 veclen) const;
```

```
  virtual void copyValueType(const void* Tsrc, void* Tdest) const;
  virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                 APT_UInt32 veclen) const;
  virtual void bindValueType(const void* Tsrc, void* Tdest) const;
  virtual void bindValueType_vec(const void* Tsrc, void* Tdest,
                                 APT_UInt32 veclen) const;

  virtual void clearValueType(void* Tdest) const;
  virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;

  virtual void setInBandNull(void* Tdest) const;

  virtual void unparse(ostream&) const;
  virtual bool isBuiltInType() const; // true for raw

  static int registerDescriptor(); // initialization support
  // first call registers this type; subsequent calls are nops

  virtual APT_FieldProtocol* protocol() const;

protected:
  virtual APT_FieldTypeDescriptor* clone() const;
  virtual void prepareForInput(void* Tval) const;
  virtual void prepareForOutput(void* Tval) const;

  virtual void parse_(const char*, APT_ParseError*);
  /* syntax: ""                 -- variable-length, aligned x1 bytes
             "5"                -- fixed-length, aligned x1 bytes
             "length=5"         -- fixed-length, aligned x1 bytes
             "max=5"            -- bounded-length, aligned x1 bytes
             "align=4"          -- variable-length, aligned x4 bytes
             "5, align=4"       -- fixed-length, aligned x4 bytes
             "length=5, align=4" -- fixed-length, aligned x4 bytes
             "max=5, align=4"   -- bounded-length, aligned x4 bytes
             "align=4, length=5" -- fixed-length, aligned x4 bytes
             "record..."        -- import schema (intact record)
  */
  virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
  virtual APT_UInt32 hash_() const;

private:
  APT_DECLARE_RTTI(APT_RawFieldDescriptor);
  APT_DECLARE_PERSISTENT(APT_RawFieldDescriptor);

  bool isFixedLength_;
  bool isBoundedLength_;

  APT_UInt32 fixedLength_;
  APT_UInt32 boundedLength_;

  int align_;
```

```
  APT_Schema* schema_;
  APT_Schema* intfSchema_;

  APT_DECLARE_NEW_AND_DELETE(APT_RawFieldDescriptor);
};

/* Cause every compilation unit that includes this header file to call
   the registerDesriptor() function for this descriptor class.  In
   this manner, we ensure that anyone who wants to use this descriptor
   class will always find that it has been registered. */
static int APT_sRegisterRaw = APT_RawFieldDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_RawField, raw, RawField);

// for back-compatibility

#ifndef APT_ACCESSOR_TEMPLATES_DEF
#define APT_ACCESSOR_TEMPLATES_DEF
template<class T> class APT_InputAccessor {};
template<class T> class APT_OutputAccessor {};
#endif

class APT_InputAccessor<APT_RawField> : public APT_InputAccessorToRawField
{
public:
  APT_InputAccessor() : APT_InputAccessorToRawField() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToRawField(comp, cur) {}
};
class APT_OutputAccessor<APT_RawField> : public APT_OutputAccessorToRawField
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToRawField() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToRawField(comp, cur) {}
};


/**** declarations for basic generic functions on raw fields ****/

class APT_RawGFEquality : public APT_GFEquality
{
  APT_DECLARE_RTTI(APT_RawGFEquality);
  APT_DECLARE_PERSISTENT(APT_RawGFEquality);
public:
  APT_RawGFEquality();
```

```
  /* default copy/assign OK */
  virtual bool isEqual(const void* Tleft, const void* Tright) const;
  virtual APT_UInt32 hash(const void* Tval) const;
  virtual APT_GenericFunction* clone() const;
  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */
};
static int APT_sRegisterRawGFEquality=
  APT_RawGFEquality::registerFunction();


class APT_RawGFComparison : public APT_GFComparison
{
  APT_DECLARE_RTTI(APT_RawGFComparison);
  APT_DECLARE_PERSISTENT(APT_RawGFComparison);
public:
  APT_RawGFComparison();
  /* default copy/assign OK */
  virtual CompareResult compare(const void* Tleft, const void* Tright) const;
  virtual APT_GenericFunction* clone() const;
  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */
};
static int APT_sRegisterRawGFComparison=
  APT_RawGFComparison::registerFunction();

class APT_RawGFPrint : public APT_GFPrint
{
  APT_DECLARE_RTTI(APT_RawGFPrint);
  APT_DECLARE_PERSISTENT(APT_RawGFPrint);
public:
  APT_RawGFPrint();
  /* default copy/assign OK */
  virtual void print(const void* Tval, ostream&) const;
  virtual APT_Status scan(const APT_String& literal, void* Tval);
  virtual APT_GenericFunction* clone() const;
  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */
};
static int APT_sRegisterRawGFPrint=
  APT_RawGFPrint::registerFunction();

#if ((defined (PROFILE)) || ((defined(__GNUC__) && ! defined(__LINUX__))))
class APT_GFIX_RawRegistration
{
public:
  static int nopForRegistration()
  { extern int APT_GFIX_RawMagicCookie; return APT_GFIX_RawMagicCookie;};
};

static int APT_GFIX_registerRaw=APT_GFIX_RawRegistration::nopForRegistration();
```

```
#endif

#endif // APT_RAW_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TYPE_STRING_H
#define APT_TYPE_STRING_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_FUNCTION_H
#include <apt_framework/type/function.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_StringDescriptor : public APT_FieldTypeDescriptor
{
  APT_DECLARE_RTTI(APT_StringDescriptor);
  APT_DECLARE_PERSISTENT(APT_StringDescriptor);

public:
  APT_StringDescriptor();

  // default copy/assign OK

  virtual APT_String accessorTypeNameFragment() const;

  bool isFixedLength() const { return isFixedLength_; }
  APT_UInt32 fixedLength() const;       // requires: isFixedLength() true

  bool isBoundedLength() const { return isBoundedLength_; }
  APT_UInt32 boundedLength() const;     // requires: isBoundedLength() true

  void setVariableLength();
  void setFixedLength(APT_UInt32);
  void setBoundedLength(APT_UInt32);
  // must be non-zero

  char padChar() const { return padChar_; }
```

```
  void setPadChar(char c) { padChar_ = c; }
  /*
     effect    Specifies the pad character used for fixed length
               strings.
  */

  virtual void* allocValueType(APT_UInt32 veclen) const;
  virtual void freeValueType(void* Tvec, APT_UInt32 veclen) const;

  virtual void copyValueType(const void* Tsrc, void* Tdest) const;
  virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                 APT_UInt32 veclen) const;
  virtual void bindValueType(const void* Tsrc, void* Tdest) const;
  virtual void bindValueType_vec(const void* Tsrc, void* Tdest,
                                 APT_UInt32 veclen) const;

  virtual void clearValueType(void* Tdest) const;
  virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;

  virtual void setInBandNull(void* Tdest) const;

  virtual void unparse(ostream&) const;
  virtual bool isBuiltInType() const; // true for string

  static int registerDescriptor(); // initialization support
  // first call registers this type; subsequent calls are nops

  virtual APT_FieldProtocol* protocol() const;

protected:
  virtual void prepareForInput(void* Tval) const;
  virtual void prepareForOutput(void* Tval) const;
  virtual APT_FieldTypeDescriptor* clone() const;

  virtual void parse_(const char*, APT_ParseError*);
  /* syntax: ""  -- variable length
             "5" -- fixed length
             "max=5" -- variable length, up to a maximum of 5
             "padchar='c'" -- specifies padChar()
  */
  virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
  virtual APT_UInt32 hash_() const;

private:
  bool isFixedLength_;
  bool isBoundedLength_;
  char padChar_;

  APT_UInt32 fixedLength_;
  APT_UInt32 boundedLength_;

  APT_DECLARE_NEW_AND_DELETE(APT_StringDescriptor);
};

/* Cause every compilation unit that includes this header file to call
```

```
   the registerDesriptor() function for this descriptor class.  In
   this manner, we ensure that anyone who wants to use this descriptor
   class will always find that it has been registered. */
static int APT_sRegisterString =
  APT_StringDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_String, string, String);

// for back-compatibility

typedef APT_InputAccessorToString APT_InputAccessorToStringField;
typedef APT_OutputAccessorToString APT_OutputAccessorToStringField;

#ifndef APT_ACCESSOR_TEMPLATES_DEF
#define APT_ACCESSOR_TEMPLATES_DEF
template<class T> class APT_InputAccessor {};
template<class T> class APT_OutputAccessor {};
#endif

class APT_InputAccessor<APT_String> :
  public APT_InputAccessorToString
{
public:
  APT_InputAccessor() : APT_InputAccessorToString() {}
  ~APT_InputAccessor();
  APT_InputAccessor(const APT_FieldSelector& comp,
                    APT_InputAccessorInterface* cur)
    : APT_InputAccessorToString(comp, cur) {}
};
class APT_OutputAccessor<APT_String> :
  public APT_OutputAccessorToString
{
public:
  APT_OutputAccessor() : APT_OutputAccessorToString() {}
  ~APT_OutputAccessor();
  APT_OutputAccessor(const APT_FieldSelector& comp,
                     APT_OutputAccessorInterface* cur)
    : APT_OutputAccessorToString(comp, cur) {}
};


/**** declarations for basic generic functions on string values ****/

class APT_StringGFEquality : public APT_GFEquality
{
  APT_DECLARE_RTTI(APT_StringGFEquality);
  APT_DECLARE_PERSISTENT(APT_StringGFEquality);
public:
  APT_StringGFEquality();
  /* default copy/assign OK */
  virtual bool isEqual(const void* Tleft, const void* Tright) const;
  virtual APT_UInt32 hash(const void* Tval) const;
  virtual APT_GenericFunction* clone() const;

  virtual void setParams(const APT_PropertyList& pl,
```

```
                        APT_PropertyList* unrecognized);
  /*
     effect    Recognizes the properties: case=sensitive (default),
               case=insensitive.
  */

  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */

private:
  bool caseSensitive_;
};
static int APT_sRegisterStringGFEquality=
  APT_StringGFEquality::registerFunction();


class APT_StringGFComparison : public APT_GFComparison
{
  APT_DECLARE_RTTI(APT_StringGFComparison);
  APT_DECLARE_PERSISTENT(APT_StringGFComparison);
public:
  APT_StringGFComparison();
  /* default copy/assign OK */
  virtual CompareResult compare(const void* Tleft, const void* Tright) const;
  virtual APT_GenericFunction* clone() const;

  virtual void setParams(const APT_PropertyList& pl,
                         APT_PropertyList* unrecognized);
  virtual APT_PropertyList params() const;
  /*
     effect    Recognizes the properties: case=sensitive (default),
               case=insensitive.
  */

  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */

private:
  bool caseSensitive_;
};
static int APT_sRegisterStringGFComparison=
  APT_StringGFComparison::registerFunction();

class APT_StringGFPrint : public APT_GFPrint
{
  APT_DECLARE_RTTI(APT_StringGFPrint);
  APT_DECLARE_PERSISTENT(APT_StringGFPrint);
public:
  APT_StringGFPrint();
  /* default copy/assign OK */
  virtual void print(const void* Tval, ostream&) const;
  virtual APT_Status scan(const APT_String& literal, void* Tval);
  virtual APT_GenericFunction* clone() const;
  static int registerFunction(); /* initialization support */
  /* first call registers this function; subsequent calls are nops */
```

```
};
static int APT_sRegisterStringGFPrint=
  APT_StringGFPrint::registerFunction();

#if ((defined (PROFILE)) || ((defined(__GNUC__) && ! defined(__LINUX__))))
class APT_GFIX_StringRegistration
{
public:
  static int nopForRegistration()
  { extern int APT_GFIX_StringMagicCookie; return APT_GFIX_StringMagicCookie;};
};

static int APT_GFIX_registerString=APT_GFIX_StringRegistration::nopForRegistration();
#endif

#endif // APT_TYPE_STRING_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_CONVERSION_H
#define APT_CONVERSION_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif

class APT_String;
class APT_ParseError;
class APT_FieldTypeDescriptor;
class APT_ErrorLog;
class APT_Lexer;

/* All type conversions occur between field value C++ types,
   notated as ST (source type) and DT (destination type).

   Type conversions are invoked by the framework to perform field
   type conversions that arise as part of the dataflow graph.  From
   the user's perspective, conversions generally occur in
   association with adapters.

   When looking for a way to convert a source type ST to a destination
   type DT, if the user has not explicitly specified a conversion in
   an adapter, then the framework looks for a default type conversion
   that goes directly from ST to DT.  The framework will not
   automatically cascade default conversions.

   If the user specifies an explicit (named) conversion in an adapter,
   then the framework will use that explicit conversion.  In addition,
   the framework will automatically cascade up to two additional
   default conversions as follows:
```

```
   Suppose that the source field type is ST and the destination field
   type is DT.  And suppose that the explicit (named) conversion
   specified by the user has a source type ST' and a destination type
   DT'.  Then the conversions used will be:

     ST  -> ST'         -- default conversion selected by framework
     ST' -> DT'         -- named conversion specified by user in adapter
     DT' -> DT          -- default conversion selected by framework

   If ST and ST' are the same, then the ST -> ST' default conversion
   is omitted; similarly, if the DT' -> DT types are the same, then
   the DT' -> DT conversion is omitted.

   A global registry keeps track of all field type conversions that
   have been defined.  Type conversions may be looked up and used by
   the client (the requisite registry and conversion functions are
   public).

   Also note that field type conversions are distinct from the
   various generic functions provided by each field type.

   TBD: conversion arity: multiple inputs fields or multiple output
   fields (to allow conversions like: c = sum(a, b) ).  Not likely.
*/

class APT_FieldConversion : public APT_Persistent
{
  APT_DECLARE_RTTI(APT_FieldConversion);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_FieldConversion);

public:
  virtual ~APT_FieldConversion();

  // default copy/assign OK

  const APT_Identifier& sourceTypeName() const { return sourceTypeName_; }
  const APT_Identifier& destTypeName() const { return destTypeName_; }
  const APT_Identifier& conversionName() const { return conversionName_; }
  /*
     effect    Returns identifying information.  See the ctor
               documentation for a description of these items.
  */

  bool isDefault() const;
  /*
     effect    Tells if this conversion was constructed as a default
               conversion.
  */

  APT_FieldConversion* own() const;
```

```
  /*
     effect    If canParameterize() is true, returns the clone() of
               this object.  Otherwise returns this object without
               cloning.
  */
  void disOwn();
  /*
     effect    If canParameterize() is true, deletes this object.
               Otherwise does nothing.
  */

  enum ParameterFlag { eImmutable=0, eCanParameterize };
protected:
  APT_FieldConversion(ParameterFlag p,
                      const char* sourceTypeName,
                      const char* destTypeName,
                      const char* conversionName="");
  /*
     effect    The ParameterFlag indicates whether this conversion
               accepts parameters, or is immutable (and therefore
               shareable).
               The sourceTypeName and destTypeName arguments identify
               the source schema type and destination schema type of
               this type conversion.  The schema type names are both
               case-insensitive.
               If a conversionName is specified, then this is an
               explicit conversion.  The conversion name is
               case-insensitive.
               If the conversionName is defaulted, then this is a
               default conversion.
     requires  All args non-null.
               The sourceTypeName and destTypeName type must be
               identifiers; they should each be the schemaTypeName() of
               a registered type descriptor (for testing, this need
               not be true).
               If non-empty, the conversionName must be an identifier.
               Each source type/destination type pair can have at most
               one default conversion.
               Explicit conversions must be uniquely named, even for
               different source type/desination type pairs.
               If this is a default conversion (empty conversionName),
               then p must be eImmutable.
  */

public:
  /**** conversion parameter management ****/

  /* If a conversion is parameterized, its class should:
     - pass ParameterFlag=eCanParameterize to the base class
       constructor
     - contain data members representing the parameter values
```

```
    - default the parameter values in its constructor
    - implement parse() and unparse()
    - in serialize(), arrange for all of the parameter values to be
      serialized.
    - optionally, provide member functions to manipulate the
      descriptor's state
  */

  bool canParameterize() const;
  /*
     effect    Indicates whether this conversion can accept parameters
               (via parse()).
     note      A non-parameterizable conversion will be shared among
               its clients, and should not be deleted.
               Parameterizable conversions are unique (heap allocated)
               and should be deleted by clients.
               In derivations, functions that modify a conversions's
               state should only be provided if canParameterize() is
               true.
  */

  APT_String ident() const;
  /*
     effect    For explicit (named) conversions, returns a string of
               the form "D=C[P](S)", where D, C, and S are the
               destination type, conversion name, and source type,
               respectively.  The [P] item is the unparse()d
               parameters for this type, if any.
               For default conversions, returns a string of the form
               "D=S".
  */

  void parse(const char* input, APT_ParseError*);
  /*
     effect    If canParameterize() is true, calls parse_().
               If canParameterize() is false, issues a parse error
               saying that this conversion doesn't take parameters.
     note      If parse_() reports a parse error, its info() is
               wrapped by the following contextual information:
               "Parsing parameters "P" for conversion "I": M",
               where I is the ident(), P is the original input to
               parse(), and M is the parse_() error's message.
     requires  input non-null
  */

protected:
  virtual void parse_(const char*, APT_ParseError*);
  /*
     effect    If this conversion is parameterized, this function
               should be overridden to parse the conversion parameters
               from the adapter definition.
```

```
              The default implementation APT_DETAIL_FATAL()s.
    throws    APT_ParseError: trouble parsing the supplied input as
                parameters for this conversion.
                If the err argument is 0, then the error is thrown;
                if err is non-null, then the error object is copied
                into the object pointed to by err.
    note      Conversion parameters appear within square brackets []
              following the conversion name.  For example:
                dest = substring [2 5] (source)
                dest = stringmap [...] (source)
              The parse() function will be called even if parameters
              are not supplied; if there are no square brackets
              following a conversion name, then parse() is called
              with the empty string as an argument.
              If empty square brackets follow the conversion name,
              then parse() is called with an empty string.
              The text between the square brackets is passed to the
              parse() routine.
    requires  For all conversions, the parameter syntax must not
              contain square brackets.
  */

public:
  virtual void unparse(ostream&) const;
  /*
    effect    If parse() is overridden, this function must be
              overridden to unparse (provide a parseable
              representation) the conversion parameters.
              The default implementation returns the empty string.
    note      The resulting string will be enclosed in square
              brackets [] by the caller.
              If the resulting string is empty, then the caller will
              omit the square brackets.
  */
  APT_String unparse() const;
  // implemented in terms of ostream form.


  /**** conversion analysis ****/

  enum Validation { eOk, eIdentical, eWarn, eError };
  virtual Validation validate(const APT_FieldTypeDescriptor* src,
                              const APT_FieldTypeDescriptor* dest,
                              APT_String* reason);
  /*
    effect    Checks that this conversion can be performed.  The
              point is that the validity of some conversions depend
              on how types are parameterized (example: var-length
              string converted to fixed-length string might pad or
              truncate), in conjunction with how the conversion
              itself is parameterized.
```

```
              The default implementation returns eOk.
   note       Overrides can assume that src and dest are of the
              appropriate derived types.
              If applicable, parse() will have been called.
   returns    eOk: conversion is fine
              eIdentical: no conversion is necessary: the two type
                descriptors are identical (as far as this conversion
                is concerned).  Note: after this interface was
                designed, the framework has been changed such that
                validate() is not called for type descriptors that
                are isEqual(); so in practice, an eIdentical return
                will rarely occur.
              eWarn: conversion has potential problem, but can be
                done; explanation is written to *reason
              eError: conversion cannot be performed; explanation is
                written to *reason
*/


/**** conversion execution ****/

/* notation: ST is source type, DT is dest type */

virtual void* allocConversionData(const APT_FieldTypeDescriptor* src,
                                  const APT_FieldTypeDescriptor* dest) const;
/*
   effect     Can be overridden to allocate and initialize a
              conversion params object that will be passed to
              convert() and convert_vec().
              The default implementation returns 0.
   note       This is intended for use in immutable conversions that
              would like to have some per-conversion object state
              (most don't need this capability).
              validate() will have been called.
              Parameterizable conversions can simply contain their
              own per-conversion object state.
              Overrides can assume that src and dest are of the
              appropriate derived types.
*/

virtual void freeConversionData(void*) const;
/*
   effect     If allocConversionData() is overridden, then this
              function should also be overridden to free the object
              allocated by allocConversionData().
              The default implementation is a nop.
*/

virtual APT_Status convert(const void* STval, void* DTval,
                           void* data) const=0;
/*
```

```
    effect    Must be overridden to perform data conversion from the
              source type to the destination type.
              The data argument is the return value of
              allocConversionData().
    note      The STval and DTval pointers point to valid (already
              constructed) instances of the source and destination C++
              value types, respectively.
    errors    A conversion failure is indicated by returning
              APT_StatusFailed; the caller will report the conversion
              failure (giving the source field name and source
              value).
              If conversion failure status is returned, the value
              written to DTval is a "don't care".
  */

protected:
  virtual APT_FieldConversion* clone() const=0;
  /*
    effect    Must be overridden to make a deep copy of this
              conversion.
              The returned object must be dynamically allocated and
              should deleted by the client.
    note      This function can be implemented in derived class D as
              follows:
                APT_FieldConversion* D::clone() const
                { return new D(*this); }
  */

private:
  APT_FieldConversion();        // required for persistence

  bool canParameterize_;
  APT_Identifier sourceTypeName_;
  APT_Identifier destTypeName_;
  APT_Identifier conversionName_;
  bool isDefault_;
};


class APT_FieldConversionRegistryRep;

class APT_FieldConversionRegistry : public APT_Persistent
{
  APT_DECLARE_PERSISTENT(APT_FieldConversionRegistry);
  APT_DECLARE_RTTI(APT_FieldConversionRegistry);

public:
  static APT_FieldConversionRegistry& get();
  /*
    effect    Returns the single global instance of the field conversion
              registry.
```

```
   note      The global registry is constructed upon first call to
             get().
*/

void addFieldConversion(APT_FieldConversion* fc);
/*
   effect    Adds the indicated field conversion object to the field
             conversion registry.
             A pointer to the argument will be retained.
   requires  fc non-null
             For a given pair of fc->sourceTypeName() and
               fc->destTypeName(), only one default conversion
               (empty fc->conversionName()) may be registered.
             For non-empty fc->conversionName(), the name must be
             globally unique.
*/

/* TBD: associate a conversion src/dest type pair (and conversion
   name) with a dynamic-load module that contains the conversion);
   load it only on demand. */

APT_FieldConversion* lookupDefault(const char* srcTypeName,
                                   const char* destTypeName) const;
/*
   effect    Locates and returns the default conversion for the
             given source and destination schema types.
             Returns null if no default conversion is registered for
             the indicated types.
   note      Schema type names are case-insensitive.
             The returned conversion is shared and must not be
             deleted nor modified.
   requires  args non-null.
*/

APT_FieldConversion* lookupExplicit(const char* conversionName) const;
/*
   effect    Locates and returns the indicated explicit (named)
             conversion.
             Returns null if no conversion is registered under the
             desired name.
   note      Conversion names are case-insensitive.
             Returns the conversion via its own() function.  The
             client must call disOwn() on the returned conversion.
   requires  args non-null.
*/

APT_FieldConversion* lookupAndParse(const char*, APT_ParseError*) const;
/*
   effect    Locates and returns the indicated explicit (named)
             conversion.
             Returns null if no conversion is registered under the
```

```
              desired name.  Case-insensitive matching.
              The string argument is expected to be of the form:
                convname
              or
                convname[params]
              If conversion params are supplied, the returned
              conversion object will have been parse()d.
              Returns null if no conversion descriptor has the given
              name (this is *not* flagged as a parse error).
    note      Returns the conversion via its own() function.  The
              client must call disOwn() on the returned conversion.
    requires  arg non-null.
  */

  int numConversions() const;
  /*
    effect    Tells how many field conversions have been added.
  */
  APT_FieldConversion* getConversion(int index) const;
  /*
    effect    Retrieves a field conversion.  The index ordering of
              conversions is unspecified.
    note      This function is not efficient.
              Returns the conversion via its own() function.  The
              client must call disOwn() on the returned conversion.
    requires  0 <= index < numConversions()
  */

private:
  APT_FieldConversionRegistry();        // clients must use get()
  ~APT_FieldConversionRegistry();

  APT_FieldConversion* lookupAndParse_(APT_Lexer&, APT_ParseError*,
                                       bool tolerateTrailingCruft=true) const;

  // prohibit copy/assign
  APT_FieldConversionRegistry(const APT_FieldConversionRegistry&);
  APT_FieldConversionRegistry& operator= (const APT_FieldConversionRegistry&);

  APT_FieldConversionRegistryRep* rep_;
};

#endif // APT_CONVERSION_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TYPE_DATE_H
#define APT_TYPE_DATE_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_DATE_H
#include <apt_util/date.h>
#endif

class APT_DateDescriptor : public APT_FieldTypeDescriptor
{
    // Don't need the destructor since there isn't anything
    // parameterizable about this class and hence there are no
    // member variables.

    APT_DECLARE_RTTI(APT_DateDescriptor);
    APT_DECLARE_PERSISTENT(APT_DateDescriptor);

  public:

    // See APT_FieldTypeDescriptor in apt_framework/type/descriptor.h
    // for documentation of these.

    APT_DateDescriptor();

    virtual void *allocValueType(APT_UInt32 veclen) const;
    virtual void freeValueType(void *Tvec, APT_UInt32 veclen) const;

    virtual void copyValueType(const void* Tsrc, void* Tdest) const;
    virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                   APT_UInt32 veclen) const;

    virtual void clearValueType(void* Tdest) const;
    virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;

    virtual void setInBandNull(void *Tdest) const;

    virtual bool isBuiltInType() const; // Will be true.
```

```
    static int registerDescriptor();
    // effect   Registers an instance of the integer descriptor on the
    //          first call; subsequent calls are nops.

    virtual APT_FieldProtocol *protocol() const;

  protected:

    virtual APT_FieldTypeDescriptor* clone() const;
};

// Make sure everyone including this file calls the registration
// function (the registration function only does something on the first
// call).  We could make this only be called once, but only at the risk of
// not having it registered in some files where people need it.
static int APT_sRegisterDate = APT_DateDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_Date, date, Date);

#endif // APT_TYPE_DATE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TYPE_DECIMAL_H
#define APT_TYPE_DECIMAL_H

#ifndef APT_DECIMAL_H
#include <apt_util/decimal.h>
#endif

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_DATE_H
#include <apt_util/decimal.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_DecimalDescriptor : public APT_FieldTypeDescriptor
{
    APT_DECLARE_RTTI(APT_DecimalDescriptor);
    APT_DECLARE_PERSISTENT(APT_DecimalDescriptor);
    // void APT_DecimalDescriptor::serialize(APT_Archive &ar, APT_UInt8 ver);

  public:

    // See APT_FieldTypeDescriptor in apt_framework/type/descriptor.h
    // for documentation of most of these.

    // Don't need the destructor since the data members don't require any
    // fancy work to destruct.

    APT_DecimalDescriptor();

    virtual void *allocValueType(APT_UInt32 veclen) const;
    virtual void freeValueType(void *Tvec, APT_UInt32 veclen) const;

    virtual void copyValueType(const void* Tsrc, void* Tdest) const;

    virtual void clearValueType(void* Tdest) const;
```

```
    virtual void setInBandNull(void *Tdest) const;

    static int registerDescriptor();
    // effect   Registers an instance of the integer descriptor on the
    //          first call; subsequent calls are nops.

    virtual APT_FieldProtocol *protocol() const;

    virtual void unparse(ostream &) const;

    int precision() const { return precision_; }
    int scale() const { return scale_; }
    // effect   Return the value of the precision and scale, respectively,
    //          that are used by this variant of the decimal type.

    void setPrecision(int);
    void setScale(int);
    // effect   Sets the value of the precision and scale, respectively,
    //          that are used by this variant of the decimal type.
    // requires precision >= 1; scale >= 0

    virtual bool isBuiltInType() const; // Will be true.

    virtual void prepareForInput(void* Tval) const;
    virtual void prepareForOutput(void* Tval) const;

  protected:
    virtual void parse_(const char* args, APT_ParseError* err);
    // Syntax: p[,s]
    //          p -- precision
    //          s -- scale (defaults to zero).

    virtual APT_FieldTypeDescriptor* clone() const;
    virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
    virtual APT_UInt32 hash_() const;

    int precision_;
    int scale_;

    APT_DECLARE_NEW_AND_DELETE(APT_DecimalDescriptor);
};

// Make sure everyone including this file calls the registration
// function (the registration function only does something on the first
// call).  We could make this only be called once, but only at the risk of
// not having it registered in some files where people need it.
static int APT_sRegisterDecimal = APT_DecimalDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_Decimal, decimal, Decimal);
```

```
#endif // APT_TYPE_DATE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_DESCRIPTOR_H
#define APT_DESCRIPTOR_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif

class APT_ParseError;
class APT_Lexer;
class APT_FieldProtocol;
class APT_FieldTypeRegistry;
class APT_UnknownFieldDescriptor;
class APT_FieldTypeRegistryRep;

class APT_FieldTypeDescriptor : public APT_Persistent
/* Abstract base class for field type descriptors.  All field type
   descriptor classes are expected to be derived directly from this
   base class.

   A type descriptor ties together the following basic concepts:

   - schema specification of field type, params
   - field representation in record transport buffer
   - field value type (built-in C++ type; or 3d party class; or
     a class specially designed to represent a field value)

   A type descriptor's fundamental role is to tie the schema-driven
   field type system to the C++ type system.  Each kind of schema
   field has a corresponding C++ type (either a built-in type, or a
   class representing the field value).
```

```
   Separately, generic functions can be provided that can be invoked
   on any field type by the client; an implementation of each generic
   function can be provided for each field type.  See
   type/function.h.

   Also separately, conversions among the various field types can be
   provided for use in adapters, or explicitly by the client.  See
   type/conversion.h.

   Finally, also as a separate activity, input and output accessor
   classes for each field type are defined.  These accessor classes
   are tied to the corresponding type descriptor as part of their
   definition.  See accessor_def.h.

   A global registry keeps track of all field type descriptors that
   have been made available.
*/
{
  APT_DECLARE_RTTI(APT_FieldTypeDescriptor);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_FieldTypeDescriptor);

public:
  virtual ~APT_FieldTypeDescriptor();

  APT_String valueTypeName() const;
  /*
     effect    Returns the type name (class or built-in C++ type) of
               the type seen by the client via the field accessor for
               this field type.
               The value type name is case-sensitive.
     note      The return type is APT_String (rather than an
               identifier) because the value-type name is
               case-sensitive.
     examples  For int32, returns APT_Int32.
               For string, returns APT_String.
  */

  const APT_Identifier& schemaTypeName() const { return schemaTypeName_; }
  /*
     effect    Returns the type name of the field type as expressed in
               an APT_Schema.
               The schema type name is case-insensitive.
     examples  For int32, returns int32.
               For string, returns string.
               For raw, returns raw (never aligned, which is
                 deprecated).
  */

  virtual APT_String accessorTypeNameFragment() const;
```

```
/*
   effect    Returns a fragment from which the accessor type names
             are (by convention) built.
             It is assumed that input accessors are named like
             APT_InputAccessorToXXX, and output accessors are named
             like APT_OutputAccessorToXXX.  This function returns
             the XXX portion.
             The base implementation returns the valueTypeName()
             with the leading APT_, if any, stripped away.  Note
             that some types, such as string field, must override
             this function (to return RawField in the raw
             case, for example).
*/

APT_UInt32 sizeofT() const { return sizeofT_; }
/*
   effect    Returns the sizeof(T), where T is the C++ type of
             this field type.
*/

APT_FieldTypeDescriptor* own() const;
/*
   effect    If canParameterize() is true, returns the clone() of
             this object.  Otherwise returns this object without
             cloning.
   requires  The returned object must not be delete'd; use disOwn()
             instead.
*/
void disOwn();
/*
   effect    If canParameterize() is true, deletes this object.
             Otherwise does nothing.
   requires  This object must have been returned from own().
*/

const APT_FieldTypeDescriptor* registeredCopy() const
{ if (isRegisteredInstance_) return this;
  else return registeredCopy_(); }
/*
   effect    Returns a copy of this type descriptor.  If this
             function is called on two type descriptor objects for
             which isEqual() is true, then the returned pointer will
             point to a shared instance that is maintained by the
             type descriptor registry.
   requires  The returned object must not be mutated.
             The returned object must not be delete'd or
             disOwn()ed.
*/

static void store(APT_Archive&,
```

```
                    const APT_FieldTypeDescriptor* registeredCopy);
  static const APT_FieldTypeDescriptor* load(APT_Archive&);
  /*
     effect    Utility routines for serializing a registered type
               desciptor pointer.
     note      Doing this correctly is extremely hard-- use these
               routines!!
  */

  bool isEqual(const APT_FieldTypeDescriptor*) const;
  /*
     effect    Tells if the given type descriptor is of the same field
               type and parameterization as this type descriptor.
  */

  APT_UInt32 hash() const;
  /*
     effect    Computes a hash of this type descriptor, taking into
               account the field type and its parameters.
  */

  bool operator== (const APT_FieldTypeDescriptor& other) const
  { return isEqual(&other); }

protected:
  enum ParameterFlag { eImmutable=0, eCanParameterize };
  APT_FieldTypeDescriptor(ParameterFlag p,
                          const char* valueTypeName,
                          APT_UInt32 sizeofT,
                          const char* schemaTypeName,
                          const char* baseValueTypeName=0);
  /*
     effect    Must be called by all derivations' constructor.
               The ParameterFlag indicates whether this descriptor
               accepts parameters, or is immutable (and therefore
               shareable).
               The valueTypeName is the type name (class or
               built-in C++ type) of the type seen by the client via
               the field accessor for this field type.  The value
               type name is case-sensitive.
               The sizeofT is the sizeof(T), where T is the value
               type.
               The schemaTypeName argument is the type name of the
               field type as expressed in an APT_Schema.  The schema
               type name is case-insensitive.
               baseValueTypeName is the name of the base field
               class (if any) of the field type represented by this
               descriptor.  The null string means the value type
               has no base class.  See the note.
     note      If a field type class is derived from another field
```

```
               type class, this can be reflected in the field type
               system.  The semantics of field type inheritance is
               described elsewhere.
               Call the base field type class BT, and the derived
               field type class DT (DT is derived from BT).
               BT and DT have distinct type descriptor classes, not
               related by inheritance (all type descriptor classes are
               derived directly from the appropriate descriptor base
               class).  However, the DT descriptor's
               baseValueTypeName can be set to BT's class name,
               thus informing the framework that DT and BT are
               related.
               Only single inheritance of field type classes is
               supported.  A related requirement is that a BT pointer
               and a DT pointer to the same instance must contain the
               same address.
    requires   The valueTypeName must be non-empty.
               sizeofT must be positive.
               The schemaTypeName must be non-empty.
               The schemaTypeName must be unique across all registered
               type descriptors.
               If non-null, the baseValueTypeName must be the
               valueTypeName of a registered type descriptor.  The
               inheritance graph must be a single-inheritance tree.
  */

  /* TBD: versioning hooks.  This will probably take two forms:

     1. A virtual (default nop in base class) that gets called before
        protocol(); the new virtual is passed the
        APT_DataSetHeaderInfo::version() number.  This form of
        versioning is appropriate for Torrent-supplied field types,
        which can be synchronized with the data set version number.

     2. A means by which a non-Torrent field type can store/retrieve
        a version number in the data set header.
  */

public:
  virtual APT_FieldProtocol* protocol() const=0;
  /*
     effect    This function is called by the framework to obtain
               information about this type's field processing
               protocol.
               See type/protocol.h.
     note      This function is called after parameters (if any) have
               been parse()d.
               This means that the protocol can be influenced by the
               type's parameter values (example: fixed-length string
               and variable-length string have different field
```

```
            processing protocols).
   requires  The returned protocol object must be dynamically
             allocated; it will be deleted by the framework.
*/

bool isFixedLength() const;
/*
   effect    Tells if this field type has a fixed-length buffer
             representation.
   note      This function calls protocol() and determines the
             answer as follows:
               APT_EmbeddedFieldProtocol: fixed length
               APT_BaseOffsetFieldProtocol: fixed length
               APT_TraversedFieldProtocol: variable length
               APT_PrefixedFieldProtocol: variable length
*/


/**** value type instance management ****/

/* The field value type is the class (or built-in C++ type) seen
   by the client.  The client uses the value type's public
   member functions (or value, if its a built-in C++ type), via an
   accessor, to interact with the field's value.

   As a notational convention, we refer to a field value type as
   "T".  So a declaration "void* Tval" is a pointer to a single
   instance of the field type; a declaration "void* Tvec" is a
   pointer to a vector of instances of the field type.

   A client must cast the void* to an appropriately typed pointer
   before accessing the field value.  For example, for int32 fields,
   the void* should be cast to APT_Int32*; for string fields, the
   void* should be cast to APT_String*.
*/

virtual void* allocValueType(APT_UInt32 veclen) const=0;
/*
   effect    Must be overridden to new a vector of
             default-constructed T instances.
             After being default-constructed, instances must be
             further intialized as appropriate for this descriptor's
             parameterization (in terms of mode, not value).
   note      The base class implementation allocates storage for
             veclen*sizeofT bytes.  This is suitable for built-in
             types such as int32.
             If veclen==1, a singleton (non-vector) allocation is
             permitted; the veclen is reliably passed to
             freeValueType() so it can use a matching deallocation
             technique.
```

```
 */
 virtual void freeValueType(void* Tvec, APT_UInt32 veclen) const=0;
 /*
    effect    Must be overridden to deallocate a vector of T,
              typically via delete[].
    note      The vec argument will have been allocated by
              allocValueType().
              The base class implementation frees storage.  This is
              suitable for built-in types such as int32.
              If veclen==1, a singleton (non-vector) deallocation is
              permitted; the deallocation technique should match that
              used in allocValueType().
 */

 virtual void copyValueType(const void* Tsrc, void* Tdest) const=0;
 virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                APT_UInt32 veclen) const;
 /*
    effect    Must be overridden to copy a value (or vector) of T
              from the src to the dest.
              The default implementation for the vector form is to
              call the value form for each vector element.
    note      Tsrc and Tdest instances are guaranteed to be the same
              type, the value type described by this type
              descriptor.
              The source and destination regions are guaranteed to
              be non-overlapping.
              The base class implementation of the value form copies
              sizeofT bytes.  This is suitable for built-in types
              such as int32.
 */

 virtual void bindValueType(const void* Tsrc, void* Tdest) const;
 virtual void bindValueType_vec(const void* Tsrc, void* Tdest,
                                APT_UInt32 veclen) const;
 /*
    effect    Like copyValueType(), except the destination is
              permitted to share the source's state.
              The default implementation is to call copyValueType().
    requires  The source state must remain stable as long as the
              destination is binded to it.
 */



 virtual void clearValueType(void* Tdest) const=0;
 virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;
 /*
    effect    Must be be overridden to clear a value (or vector) of T
              to some default value (to prevent output value leakage
              from one output record to the next).
```

```
              The default implementation for the vector form is to
              call the value form for each vector element.
     note     The base class implementation of the value form zeroes
              sizeofT bytes.  This is suitable for built-in types
              such as int32.
  */

  void serializeValueType(APT_Archive& ar, void* Tval) const;
  void serializeValueType_vec(APT_Archive& ar, void* Tvec,
                              APT_UInt32 veclen) const;
  /*
     effect   Perform archive serialization of the indicated value
              (or vector) of T.
     note     These functions are shockingly inefficient.
              Serialization/deserialization must be accomplished
              using the same function; serializeValueType() on a
              T field is not necessarily the same as "ar || T".
  */

  virtual void setInBandNull(void* Tdest) const=0;
  /*
     effect   Sets the value to its type's in-band null value.
  */

//protected:
public: // framework use only
  virtual void prepareForInput(void* Tval) const;
  virtual void prepareForOutput(void* Tval) const;
  virtual void prepareForInput_vec(void* Tvec, APT_UInt32 veclen) const;
  virtual void prepareForOutput_vec(void* Tvec, APT_UInt32 veclen) const;
  /*
     effect   Called for newly-allocated T instances, to indicate
              whether they correspond to input fields or output
              fields.
              The default implementation of the value form is a nop.
              The default implementation of the vector form is to
              call the value form for each vector element.
     note     Base/offset and prefixed value types typically use this
              notification to free their private buffer storage,
              paving the way for the framework to manage their
              base/offset (or bufferp/length) state.
              The implementation of these functions typically has a
              side-effect of clearing the value(s).  The framework
              will arange for value(s) to be saved and restored as
              appropriate.
              After a T instance has been prepared via one of these
              functions, but before its base/offset or bufferp/length
              has been initialized by the framework, it must be
              possible to safely destroy the T instance.
  */
```

```
  /**** type parameter management ****/

  /* If a type is parameterized, its descriptor class should:
     - pass ParameterFlag=eCanParameterize to the base class
       constructor
     - contain data members representing the parameter values
     - default the parameter values in its constructor
     - implement parse() and unparse()
     - implement isEqual_() and hash_()
     - in serialize(), arrange for all of the parameter values to be
       serialized.
     - optionally, provide non-const member functions to manipulate
       the descriptor's state
     - optionally, provide a non-default constructor that initializes
       the descriptor's state
  */

public:
  bool canParameterize() const;
  /*
     effect    Indicates whether this type can accept type parameters
               (via parse()).
     note      In derivations, functions that modify a descriptor's
               state should only be provided if canParameterize() is
               true.
  */

  void parse(const char* input, APT_ParseError*);
  /*
     effect    If canParameterize() is true, calls parse_().
               If canParameterize() is false, issues a parse error
               saying that this type doesn't take parameters.
     note      If parse_() reports a parse error, its info() is
               wrapped by the following contextual information:
               "Parsing parameters "P" for schema type "T": M", where
               T is the schema type name, P is the original input to
               parse(), and M is the parse_() error's message.
     requires  input non-null
  */

protected:
  virtual void parse_(const char*, APT_ParseError*);
  /*
     effect    If this type descriptor is parameterized, this function
               should be overridden to parse the type parameters from
               the schema type definition.
               The default implementation APT_DETAIL_FATAL()s.
     throws    APT_ParseError: trouble parsing the supplied input as
```

```
                parameters for this type descriptor.
                If the err argument is 0, then the error is thrown;
                if err is non-null, then the error object is copied
                into the object pointed to by err.
    note      Type parameters appear within square brackets []
              following the type name.  For example:
                string[]
                string[12]
                string[maxlength=200]
                raw[4]
                raw[length=4,align=2]
              For parameterizable types, the parse() function will be
              called even if parameters are not supplied (in which
              case the empty string is passed).
              If empty brackets follow the type name, then parse() is
              called with an empty string.
              The text between the square brackets is passed to the
              parse() routine.
    requires  For all types, the parameter syntax must not contain
              square brackets, unless the parameter is a record
              schema (beginning with the word 'record').
  */

  virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
  /*
    effect    Must be overridden in parameterized descriptors to
              compare the given descriptor's parameter values to this
              descriptor's.
              The default implementation APT_DETAIL_FATAL()s.
              This function will not be called for non-parameterized
              type descriptors.
    note      The argument is guaranteed to have the same dynamic
              class type as this descriptor.
  */

  virtual APT_UInt32 hash_() const;
  /*
    effect    Must be overridden in parameterized descriptors to
              compute a hash function of this descriptor's parameter
              values.
              The default implementation APT_DETAIL_FATAL()s.
              This function will not be called for non-parameterized
              type descriptors.
    requires  If two descriptors have isEqual_() true, they must hash
              to the same value.
  */

public:
  virtual void unparse(ostream&) const;
  /*
```

```
    effect    If parse() is overridden, this function must be
              overridden to unparse (provide a parseable
              representation) the type parameters.
              The default implementation returns the empty string.
    note      The resulting string will be enclosed in square
              brackets [] by the caller.
              If the resulting string is empty, then the caller will
              omit the square brackets.
  */
  APT_String unparse() const;
  // implemented in terms of ostream form.

  void fullUnparse(ostream&) const;
  APT_String fullUnparse() const;
  /*
    effect    Provides the schema representation of this type,
              including the schemaTypeName(), and unparse()d type
              parameters (if any).
  */

  virtual bool isBuiltInType() const;
  /*
    effect    Indicates whether this field type is built-in, meaning
              it is provided by APT as part of the base system (will
              always be present).
              The default implementation returns false; only
              APT-provided field types should override this to return
              true, and only if the field type will always be
              present.
  */

protected:
  virtual APT_FieldTypeDescriptor* clone() const=0;
  /*
    effect    Must be overridden to make a deep copy of this
              descriptor.
              The returned object must be dynamically allocated and
              should deleted by the client.
    note      This function can be implemented in derived class D as
              follows:
                APT_FieldTypeDescriptor* D::clone() const
                { return new D(*this); }
  */

  // allow copy/assign to be generated for derived classes
  APT_FieldTypeDescriptor(const APT_FieldTypeDescriptor&);

  APT_FieldTypeDescriptor& operator= (const APT_FieldTypeDescriptor&);
  // only type descriptors of the same type may be assigned
```

```
private:
  friend class APT_FieldTypeDescriptor_UT; // back-door for testing
  friend class APT_FieldTypeRegistryRep;

  APT_FieldTypeDescriptor();    // required for persistence

  const APT_FieldTypeDescriptor* registeredCopy_() const;

  bool canParameterize_;
  bool isRegisteredInstance_;
  APT_String valueTypeName_;
  APT_Identifier schemaTypeName_;
  APT_UInt32 sizeofT_;
  bool hasBase_;
  APT_String* baseValueTypeName_;
};


class APT_FieldTypeRegistry
{
public:
  static APT_FieldTypeRegistry& get();
  /*
     effect    Returns the single global instance of the field type
               registry.
     note      The global registry is constructed upon first call to
               get().
  */

  void addFieldType(APT_FieldTypeDescriptor* td);
  /*
     effect    Adds the indicated type descriptor object to the field
               type registry.
               A pointer to the argument will be retained.
     requires  td non-null
               td->schemaTypeName() must not be the same as any
                 previously added type descriptor's schema type
                 name (this comparison is case-insensitive).
  */

  /* TBD: function to associate a schema type name with a dynamic-load
     module that contains the type descriptor (and related
     conversions, etc.) for the type; load it only on demand. */

  APT_FieldTypeDescriptor* lookupBySchemaTypeName(const char*) const;
  /*
     effect    Locates and returns the type descriptor with the given
               schema type name.  Case-insensitive matching.
               Returns null if no type descriptor has the given
               schema type name.
```

```
    note      Returns the descriptor via its own() function.  The
              client must call disOwn() on the returned descriptor.
    requires  arg non-null.
  */

  const APT_FieldTypeDescriptor* lookupAndParse(const char*,
                                                APT_ParseError*) const;
  /*
    effect    Locates and returns the type descriptor with the given
              schema type name.  Case-insensitive matching.
              The string argument is expected to be of the form:
                typename
              or
                typename[params]
              If the type accepts parameters, the returned type
              descriptor object will have been parse()d even if no
              [params] were supplied.
              Returns null if no type descriptor has the given
              schema type name (this is *not* flagged as a parse
              error).
    note      Returns the descriptor via the registeredCopy()
              function.
    requires  arg non-null.
  */

  int numTypeDescriptors() const;
  /*
    effect    Tells how many type descriptors have been added.
    note      This set is populated according to schemaTypeName();
              differently parameterized descriptor instances of the
              same type are not reflected in this set.
  */
  APT_FieldTypeDescriptor* getDescriptor(int index) const;
  /*
    effect    Retrieves a type descriptor.  The index ordering of
              type descriptors is unspecified.
    note      This function is not efficient.
              Returns the descriptor via its own() function.  The
              client must call disOwn() on the returned descriptor.
    requires  0 <= index < numTypeDescriptors()
  */

  const APT_FieldTypeDescriptor* lookupAndParse_(APT_Lexer&, APT_ParseError*,
                                  bool tolerateTrailingCruft=true) const;

private:
  friend class APT_FieldTypeDescriptor;

  APT_FieldTypeRegistry();      // clients must use get()
  ~APT_FieldTypeRegistry();
```

```
  // prohibit copy/assign
  APT_FieldTypeRegistry(const APT_FieldTypeRegistry&);
  APT_FieldTypeRegistry& operator= (const APT_FieldTypeRegistry&);

  const APT_FieldTypeDescriptor* registerDesc(const APT_FieldTypeDescriptor*);

  APT_FieldTypeRegistryRep* rep_;
};

#endif // APT_DESCRIPTOR_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_FUNCTION_H
#define APT_FUNCTION_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif


class APT_FieldTypeDescriptor;
class APT_String;
class APT_PropertyList;
class APT_GenericFunctionRegistry;

class APT_GenericFunction : public APT_Persistent
/* Abstract base class from which all generic functions are derived.

   A two-level derivation hierarchy is used.  The first level
   derivation defines a generic function interface; each interface
   class defines one or more public pure virtual member functions that
   define the actions that characterize the particular generic
   function interface being defined.

   The second level derivation provides one or more implementations
   for each generic function interface.

   The following pre-defined first level derived classes (function
   interfaces) are provided: APT_GFEquality, APT_GFComparison,
```

```
   APT_GFPrint.  Others can be added as needed either by Torrent or by
   other parties.  For example, generic functions related to data
   format translation (import/export) are defined elsewhere.

   When a client performs a generic function lookup operation, the
   client is given a base APT_GenericFunction pointer.  To use this
   pointer, the client performs a down-cast to the appropriate
   first-level derived class (exposing the pure virtual function(s)
   defining the generic function's interface), and calls the desired
   pure virtual functions to parameterize the generic function and to
   perform the generic operation.
*/
{
  APT_DECLARE_RTTI(APT_GenericFunction);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_GenericFunction);

protected:
  APT_GenericFunction(const char* schemaTypeName,
                      const char* interfaceName,
                      const char* implementationName);
  /*
     effect    The schemaTypeName argument identifies the schema
               type for which this generic function is defined.  The
               schema type name is case-insensitive.
               The interfaceName uniquely identifies this generic
               function interface among all interfaces for the given
               schema type.  The interface name is case-insensitive.
               The implementationName uniquely identifies this generic
               function implementation among all implementations for
               the given interface and schema type.  The implementation
               name is case-insensitive.
               If an implementationName is specified, then this
               implementation must be selected explicitly by name.
               If the implementationName is the empty string, then
               this is the default implementation for the given
               interface and schema type.
     note      Function interfaces are named within the scope of a
               schema type; each schema type has its own set of
               named generic function interfaces.
               Similarly, function implementations are named within
               the scope of a function interface and schema type.
     requires  All args non-null.
               The schemaType must be the schemaTypeName() of a
               registered type descriptor.
               The interfaceName must be a non-empty identifier.
  */

public:
  virtual ~APT_GenericFunction();

  // define non-inline copy constructor.  Note that this constructor
```

```
  // is in place solely for the purposes of the "clone()" virtual
  // method.
  APT_GenericFunction(const APT_GenericFunction&);

  // Forbid assignment
private:
  APT_GenericFunction& operator= (const APT_GenericFunction&);

public:
  APT_Identifier schemaTypeName() const;
  APT_Identifier interfaceName() const;
  APT_Identifier implementationName() const;
  /*
     effect    Returns identifying information.  See the ctor
               documentation for a description of these items.
  */

  APT_String ident() const;
  /*
     effect    Returns a string identifying this generic function in
               terms of its schemaTypeName(), interfaceName(), and
               implementationName().
  */

  bool isDefault() const;
  /*
     effect    Tells if this function object was constructed as a
               default implementation (see ctor).
  */

  virtual void setArgType(const APT_FieldTypeDescriptor* desc);
  /*
     effect    Derivations can override this function to pick up
               parameter information from the supplied desc.
               The intention is that desc is the type descriptor of
               the field values upon which this generic function will
               be called.
               Default implementation is a nop.
     note      This function is a protocol *suggestion*.  Note that it
               is up to the user of a generic function to actually
               call this function, so that derivations (if they care)
               can pick up type parameterization information.
               Derivations will typically require the desc to be of the
               particular field type handled by the derivation.
               Derivations should copy any needed information from
               desc; derivations should *not* just cache a copy of the
               desc pointer (of course, a derivation can keep its
               own() copy of the desc pointer if it wishes).
  */

  /* parameterization support: based on property lists */
```

```
  virtual void setParams(const APT_PropertyList& pl,
                         APT_PropertyList* unrecognized);
  /*
     effect    Should be overridden to interpret the supplied property
               list as parameters describing how this generic function
               should operate.
               If non-null, the unrecognized argument should be set to
               contain properties, if any, that were not recognized by
               this generic function, or are not set to valid values.
               The default implementation sets unrecognized to all of
               the input properties (no properties recognized by
               default).
     note      This function is a protocol *suggestion*.  Note that it
               is up to the user of a generic function to actually
               define function parameters as a property list, and to
               call this function.  If a particular kind of generic
               function wants to be parameterized in a different
               fashion, that's OK.
               At the base class, the effect is unspecified if one or
               more properties are unrecognized.
               Derivations may choose to provide clearer guidance in
               such cases (for example, the import/export generic
               functions accept all recognized properties while
               cleanly rejecting unrecognized (or wrongly formatted)
               properties.
               Note that the LSP (Liskov Substitution Principle)
               permits this kind of interface refinement (a derived
               instance will work in all base contexts).
  */

  virtual APT_GenericFunction* clone() const=0;
  /*
     effect    Must be overridden to make a deep copy of this
               function object.
               The returned object must be dynamically allocated and
               should deleted by the client.
     note      This function can be implemented in derived class D as
               follows:
                 APT_GenericFunction* D::clone() const
                 { return new D(*this); }
  */

protected:
  APT_GenericFunction();        // Required for persistence.

private:
  APT_Identifier schemaTypeName_;
  APT_Identifier interfaceName_;
  APT_Identifier implementationName_;
};
```

```
class APT_GenericFunctionRegistryRep;

class APT_GenericFunctionRegistry : public APT_Persistent
{
  APT_DECLARE_PERSISTENT(APT_GenericFunctionRegistry);
  APT_DECLARE_RTTI(APT_GenericFunctionRegistry);

public:
  static APT_GenericFunctionRegistry& get();
  /*
     effect    Returns the single global instance of the generic
               function registry.
     note      The global registry is constructed upon first call to
               get().
  */

  void addGenericFunction(APT_GenericFunction* gf);
  /*
     effect    Adds the indicated function group object to the generic
               function registry.
               A pointer to the argument will be retained.
     requires  gf non-null
               For a given gf->schemaTypeName() and
                 gf->interfaceName(), the gf->implementationName()
                 must be unique.
  */

  /* TBD: associate a schema type/implementation name/interface name
     with a dynamic-load module that contains the function); load it
     only on demand. */

  APT_GenericFunction* lookupDefault(const char* schemaType,
                                     const char* interfaceName) const;
  /*
     effect    Locates and returns the default generic function
               implementation for the indicated schema type and
               interface name.
               Returns null if no default generic function
               implementation is registered for the indicated type and
               interface.
     note      Schema type name is case-insensitive.
               Interface name is case-insensitive.
               The returned function object is dynamically allocated;
               the client is responsible for deleting it.
     requires  args non-null.
  */

  APT_GenericFunction* lookupExplicit(const char* schemaType,
                                      const char* interfaceName,
                                      const char* implementationName) const;
```

```
  /*
     effect    Locates and returns the specified generic function
               implementation for the indicated schema type and
               interface name.
               Returns null if no generic function implementation is
               registered with the given name for the indicated type
               and  interface.
     note      Schema type name is case-insensitive.
               Interface name is case-insensitive.
               Implementation name is case-insensitive.
               The returned function object is dynamically allocated;
               the client is responsible for deleting it.
     requires  args non-null.
  */

  int numGenericFunctions() const;
  /*
     effect    Tells how many functions have been added.
  */
  APT_GenericFunction* getGenericFunction(int index) const;
  /*
     effect    Retrieves a generic function group.  The index ordering of
               groups is unspecified.
     note      The returned function object is dynamically allocated;
               the client is responsible for deleting it.
     requires  0 <= index < numGenericFunctions()
  */

private:
  APT_GenericFunctionRegistry();        // clients must use get()
  ~APT_GenericFunctionRegistry();

  // prohibit copy/assign
  APT_GenericFunctionRegistry(const APT_GenericFunctionRegistry&);
  APT_GenericFunctionRegistry& operator=(const APT_GenericFunctionRegistry&);

  APT_GenericFunctionRegistryRep* rep_;
};


// pre-defined generic function interfaces

class APT_GFEquality : public APT_GenericFunction
{
  APT_DECLARE_RTTI(APT_GFEquality);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_GFEquality);

protected:
  APT_GFEquality(const char* schemaTypeName,
                 const char* implementationName);
  // passes args through to base ctor; interfaceName is "equality"
```

```
  // default copy/dtor OK; assign prohibited in base class.
  // copy is only for purposes of clone() method.

public:
  /* In the functions below, T refers to the field value type (C++
     built-in type or class) that the client sees when manipulating a
     field value via an accessor. */

  virtual bool isEqual(const void* Tleft, const void* Tright) const=0;
  /*
     effect    Must be overridden to determine if the two T instances
               are equal.
     note      Tleft and Tright are guaranteed to be the same type,
               the schema type named by schemaTypeName().
  */
  virtual APT_UInt32 hash(const void* Tval) const=0;
  /*
     effect    Must be overridden to return a hash of the supplied T
               instance.
     note      Tval is guaranteed to be an instance of the schema
               type named by schemaTypeName().
  */

private:
  APT_GFEquality();     // required for persistence
};

class APT_GFComparison : public APT_GenericFunction
{
  APT_DECLARE_RTTI(APT_GFComparison);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_GFComparison);

protected:
  APT_GFComparison(const char* schemaTypeName,
                   const char* implementationName);
  // passes args through to base ctor; interfaceName is "comparison"

  // default copy/dtor OK; assign prohibited in base class.
  // copy is only for purposes of clone() method.

public:
  /* In the function below, T refers to the field value type (C++
     built-in type or class) that the client sees when manipulating a
     field value via an accessor. */

  enum CompareResult { eLessThan=-1, eEqual=0, eGreaterThan=1 };
  virtual CompareResult compare(const void* Tleft, const void* Tright) const=0;
  /*
     effect    Must be overridden to compare the two T instances.
     note      Tleft and Tright are guaranteed to be the same type,
```

```
               the schema type named by schemaTypeName().
  */

private:
  APT_GFComparison();   // required for persistence
};

class APT_GFPrint : public APT_GenericFunction
/* Debug-quality field printing service.  This is not intended for use
   as part of an exporter, nor as part of a report generator.

   Also provides a function for turning string literals into a value
   of the field type.
*/
{
  APT_DECLARE_RTTI(APT_GFPrint);
  APT_DECLARE_ABSTRACT_PERSISTENT(APT_GFPrint);

protected:
  APT_GFPrint(const char* schemaTypeName,
              const char* implementationName);
  // passes args through to base ctor; interfaceName is "print"

  // default copy/dtor OK; assign prohibited in base class.
  // copy is only for purposes of clone() method.

public:
  /* In the function below, T refers to the field value type (C++
     built-in type or class) that the client sees when manipulating a
     field value via an accessor. */

  virtual void print(const void* Tval, ostream&) const=0;
  /*
     effect    Must be overriden to print a T instance to the given
               ostream.
  */
  APT_String print(const void* Tval) const;
  // implemented in terms of ostream form.

  virtual APT_Status scan(const APT_String& literal, void* Tval)=0;
  /*
     effect    Must be overridden to scan a T instance from the
               supplied literal string representation.
     returns   Whether the scan was successful or not.  If unsuccessful,
               the value written to Tval is unspecified.
  */

private:
  APT_GFPrint();        // required for persistence
};
```

```
#endif // APT_FUNCTION_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_PROTOCOL_H
#define APT_PROTOCOL_H

#ifndef APT_SCHEMA_H
#include <apt_framework/schema.h>
#endif

#ifndef APT_FIELDSEL_H
#include <apt_framework/fieldsel.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_CONDITION_H
#include <apt_util/condition.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

#include <string.h>

class APT_OutputAccessorBase;
class APT_TagAccessor;
class APT_DataSetRep;


class APT_FieldProtocol
/* Abstract base class for finite set (APT-defined) of field type
   processing protocols.  No user derivations are expected from this
   base class (however, there are sub-base classes from which user
   derivations are expected-- APT_TraversedFieldProtocol).

   Field type processing protocols tell the framework how to map
   between a C++ field value type and the record buffer
   representation of the field value.

   The protocol information is used by the framework when traversing
   input records and when assembling output records.  The user of a
   field type need not know anything about the protocol.
*/
{
  APT_DECLARE_RTTI(APT_FieldProtocol);
```

```
public:
  virtual ~APT_FieldProtocol();

protected:
  APT_FieldProtocol();

private:
  // prohibit copying
  APT_FieldProtocol(const APT_FieldProtocol&);
  APT_FieldProtocol& operator= (const APT_FieldProtocol&);
};

class APT_EmbeddedFieldProtocol : public APT_FieldProtocol
/* Embedded field type instances can be contained within the record
   buffer.  Such types:

   - must contain no pointers,
   - must have no virtual functions,
   - must have a value that is independent of the instance's address,
   - must have constructor(s) with no side effects other than setting
     the instance's value,
   - must be bitwise-copyable.

   Third-party classes may be used as embedded field types if they
   satisfy the above requriements.

   The record buffer representation of embedded field types is a fixed
   number of contiguous bytes, characterized by a length and a
   starting alignment.

   In the absence of a conversion, user accessors to such fields
   "point" directly into the record buffer.

   This is a concrete class, no derivations are expected.

   Examples: int32, dfloat
*/
{
public:
  APT_EmbeddedFieldProtocol(APT_UInt32 fixedLength, int align);
  /*
     effect    The fixedLength arg specifies the number of contiguous
               bytes to be used in the buffer representation for the
               field type.
               The align arg specifies the required degree of
               alignment to be maintained for the first byte of the
               buffer representation for the field type.
     requires  align must be a power of 2 (1, 2, 4, 8, etc.)
               fixedLength must be a positive integer multiple of the
               align.
  */

  APT_UInt32 fixedLength() const { return fixedLength_; }
  int align() const { return align_; }
```

```
private:
  APT_UInt32 fixedLength_;
  int align_;

  APT_DECLARE_RTTI(APT_EmbeddedFieldProtocol);
#ifndef __LINUX__
  APT_DECLARE_NEW_AND_DELETE(APT_EmbeddedFieldProtocol);
#endif
};

/* macro to yield the offset, in bytes, of a struct/class member from
   the beginning of that struct/class */
#define APT_OFFSET_OF(T, m) (APT_UInt32) (unsigned long) (char*) &((T*)0)->m

class APT_BaseOffsetFieldProtocol : public APT_FieldProtocol
/* Base/offset field class instances are heap-allocated, and contain a
   base pointer and offset that is used by the field class
   implementation to find its buffer storage.

   Initially, a base/offset instance allocates its own buffer storage,
   and arranges for its basep/offset members to refer to it.  However,
   when the framework creates base/offset instances for use in
   conjunction with accessors, the instances' own buffer storage is
   generally freed and replaced by framework-supplied storage (and the
   framework takes over management of the instance's basep and offset
   members.

   Generally, only classes designed for use as ORCHESTRATE field
   values can use this protocol.

   The record buffer representation of base/offset field classes is
   fixed length.

   A user accessor to such a field "points" to the allocated value
   instance, not into the record buffer.

   This is a concrete class, no derivations are expected.

   Examples: fixed-length string, fixed-length raw
*/
{
public:
  APT_BaseOffsetFieldProtocol(APT_UInt32 fixedLength, int align,
                              APT_UInt32 offsetOfBasep,
                              APT_UInt32 offsetOfOffset);
  /*
     effect    The fixedLength arg specifies the number of contiguous
               bytes to be used in the buffer representation for the
               field type.
               The align arg specifies the required degree of
               alignment to be maintained for the first byte of the
               buffer representation for the field type.
               The offsetOfBasep arg specifies the offset of the
                 char** basep_
               data member of the T field value type.
```

```
              The offsetOfOffset arg specifies the offset of the
                APT_UInt32 offset_
              data member of the T field value type.
    requires  align must be a power of 2 (1, 2, 4, 8, etc.)
              fixedLength must be a positive integer multiple of the
              align.
              The field value type (T) class is required to have
              the data members:
                char** basep_;
                APT_UInt32 offset_;
              These will always be manipulated by the framework to
              point to the T instance's corresponding state storage, as
              follows:
                state_addr == *basep_ + offset
  */

  APT_UInt32 fixedLength() const { return fixedLength_; }
  int align() const { return align_; }
  APT_UInt32 offsetOfBasep() const { return offsetOfBasep_; }
  APT_UInt32 offsetOfOffset() const { return offsetOfOffset_; }

private:
  APT_UInt32 fixedLength_;
  int align_;
  APT_UInt32 offsetOfBasep_;
  APT_UInt32 offsetOfOffset_;

  APT_DECLARE_RTTI(APT_BaseOffsetFieldProtocol);
#ifndef __LINUX__
  APT_DECLARE_NEW_AND_DELETE(APT_BaseOffsetFieldProtocol);
#endif
};

class APT_TraversedFieldProtocol : public APT_FieldProtocol
/* Only APT-supplied field types may use the traversed protocol.

   Traversed field class instances are heap-allocated and always
   manage their own internal storage.

   The record buffer representation of traversed field classes is
   determined by functions overridden in a derivation from this base
   class.  The purpose is to support variable-length buffer
   representations that employ a length encoding other than the
   prefix byte(s) method of APT_PrefixedFieldProtocol.

   A user accessor to such a field "points" to the allocated value
   instance, not into the record buffer.

   This is an abstract base class.  With the exception of
   APT_PrefixedFieldProtocol, only APT-supplied field types may use
   the traversed protocol.

   TBD: fixed-length variant?  Framework could sometimes exploit this,
   and would allow for type-naive handling of such fields.
```

```
   TBD: variant that generates prefix length, but still calls traversal
   virtuals?  Again, nicely allows for type-naive handling.

   Examples: null-terminated string, decimal (?)
*/
{
  APT_DECLARE_RTTI(APT_TraversedFieldProtocol);

  // default ctor OK

public:
  virtual void skip(const char* &src) const=0;
  virtual void skip_vec(const char* &src, APT_UInt32 veclen) const;
  /*
     effect    Called within getRecord().
               Must be overridden to process the field instance's
               buffer representation.
               The src pointer initially points to the beginning of
               the field; upon return, it must have been advanced to
               point to the byte following the final byte of the
               field.
               For the vector form, the veclen parameter indicates the
               number of field instances that occur (successively) in
               the buffer.
               The default implementation of the vector form is
               implemented in terms of the value form.
  */

  virtual void get(const char* &src, void* Tval) const=0;
  virtual void get_vec(const char* &src, void* Tvec, APT_UInt32 veclen,
                       APT_UInt32 sizeofT) const;
  /*
     effect    Called within getRecord() (and transferFromBuffer()).
               Must be overridden to process the field instance's
               buffer representation.
               The src pointer initially points to the beginning of
               the field; upon return, it must have been advanced to
               point to the byte following the final byte of the
               field.
               For the vector form, the veclen parameter indicates the
               number of field instances that occur (successively) in
               the buffer.
               The default implementation of the vector form is
               implemented in terms of the value form.
               Typically, this function updates a T value to tie the
               instance (in a T-dependent manner) to its buffer
               representation.
  */

  virtual void getPacked(const char* &src, void* Tval) const=0;
  virtual void getPacked_vec(const char* &src, void* Tvec, APT_UInt32 veclen,
                             APT_UInt32 sizeofT) const;
  /*
     effect    Like get() and get_vec(), except the buffer
               representation is packed (no alignment padding).  This
```

```
            is used in the transfer to/from buffer logic.
*/

virtual APT_UInt32 bufLength(const void* Tval) const=0;
virtual APT_UInt32 bufLength_vec(const void* Tvec, APT_UInt32 veclen,
                           APT_UInt32 sizeofT) const;
/*
   effect    Called within putRecord() (and getTransferBufferSize()).
             Must be overridden to return the length of the field
             instance's buffer representation.
             For the vector form, the veclen parameter indicates the
             number of field instances to be stored (successively)
             in the buffer.
             The default implementation of the vector form is
             implemented in terms of the value form.
   note      An slight overestimate of the length of the buffer
             representation is acceptable; however, large
             overestimates introduce penalties and should be
             avoided.
*/

virtual APT_UInt32 bufLengthPacked(const void* Tval) const=0;
virtual APT_UInt32 bufLengthPacked_vec(const void* Tvec, APT_UInt32 veclen,
                                 APT_UInt32 sizeofT) const;
/*
   effect    Like bufLength() and bufLength_vec(), except the buffer
             representation is packed (no alignment padding).  This
             is used in the transfer to/from buffer logic.
*/

virtual void put(const void* Tval, char* &dest) const=0;
virtual void put_vec(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                     char* &dest) const;
/*
   effect    Called within putRecord() (and transferToBuffer()).
             Must be overridden to generate the field instance's
             buffer representation.
             The dest pointer initially points to the beginning of
             the field; upon return, it must have been advanced to
             point to the byte following the final byte of the field
             (and the bytes of the field must have been written with
             the field's value in its buffer representation).
             For the vector form, the veclen parameter indicates the
             number of field instances to be stored (successively)
             in the buffer.
             The default implementation of the vector form is
             implemented in terms of the value form.
*/

virtual void putPacked(const void* Tval, char* &dest) const=0;
virtual void putPacked_vec(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                           char* &dest) const;
/*
   effect    Like put() and put_vec(), except the buffer
             representation is packed (no alignment padding).  This
```

```
              is used in the transfer to/from buffer logic.
  */

private:
  friend class APT_TraversedFieldProtocol_UT;
  friend class APT_DataSetRep;
};

class APT_PrefixedFieldProtocol : public APT_TraversedFieldProtocol
/* Prefixed field class instances are heap-allocated, and contain a
   buffer pointer and length that is used by the field class
   implementation to find its buffer state.  Generally, only classes
   designed for use as ORCHESTRATE field values can use this
   protocol.

   Initially, a prefixed instance allocates its own buffer storage,
   and arranges for its bufferp/length members to refer to it.
   However, when the framework creates prefixed instances for use in
   conjunction with input accessors, the instances' own buffer storage
   is generally freed and replaced by framework-supplied storage (and
   the framework takes over management of the instance's bufferp and
   length members.

   The record buffer representation of prefixed field classes is
   variable-length, with a prefix of 1 or 4 bytes (managed by the
   framework) to represent the field length.

   A user accessor to such a field "points" to the allocated value
   instance, not into the record buffer.

   This is a concrete class, no derivations are expected.

   Examples: variable-length string, variable-length raw
*/
{
public:
  APT_PrefixedFieldProtocol(int align,
                            APT_UInt32 offsetOfBufferp, APT_UInt32 offsetOfLength,
                            bool manageStorage=false,
                            APT_UInt32 offsetOfFlag=0, int bitNumOfFlag=0,
                            bool invertFlag=false,
                            void (*delfunc)(void* Tval)=0);
  /*
     effect    The align arg specifies the required degree of
               alignment to be maintained for the first byte of the
               buffer representation for the field type (after the
               length byte(s), which are not aligned).
               The offsetOfBufferp arg specifies the offset of the
                 void* bufferp_;
               data member of the T field value type.
               The offsetOfLength arg specifies the offset of the
                 APT_UInt32 length_;
               data member of the T field value type.
               The manageStorageFlag, if true, specifies that the
               prefixed protocol should free the T instance's
```

```
            buffer state (via the supplied delfunc) if a flag bit
            is set before setting the buffer pointer to a new
            value.
   requires  align must be a power of 2 (1, 2, 4, 8, etc.)
            The field value type (T) class is required to have
            the data members:
              void* bufferp_;
              APT_UInt32 length_;
            These will be manipulated by the framework to reflect
            the T instance's corresponding buffer storage and
            length.  On input, bufferp_ will be pointed into the
            buffer at the field's start (after the length byte(s)
            and padding, if any), and length_ will be written to
            reflect the field length.  On output, length_ bytes
            pointed to by bufferp_ will be copied into the output
            buffer.
   note     The field's length is the "payload" length; the length
            byte(s) and/or alignment byte(s) are not counted as
            part of the length.
*/

int align() const { return alignMask_+1; }
APT_UInt32 offsetOfBufferp() const { return offsetOfBufferp_; }
APT_UInt32 offsetOfLength() const { return offsetOfLength_; }

// implementations of base protocol
virtual void skip(const char* &src) const;
virtual void skip_vec(const char* &src, APT_UInt32 veclen) const;

virtual void get(const char* &src, void* Tval) const;
virtual void get_vec(const char* &src, void* Tvec, APT_UInt32 veclen,
                     APT_UInt32 sizeofT) const;

virtual void getPacked(const char* &src, void* Tval) const;
virtual void getPacked_vec(const char* &src, void* Tvec, APT_UInt32 veclen,
                           APT_UInt32 sizeofT) const;

virtual APT_UInt32 bufLength(const void* Tval) const;
virtual APT_UInt32 bufLength_vec(const void* Tvec, APT_UInt32 veclen,
                             APT_UInt32 sizeofT) const;

virtual APT_UInt32 bufLengthPacked(const void* Tval) const;
virtual APT_UInt32 bufLengthPacked_vec(const void* Tvec, APT_UInt32 veclen,
                                   APT_UInt32 sizeofT) const;

virtual void put(const void* Tval, char* &dest) const;
virtual void put_vec(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                     char* &dest) const;

virtual void putPacked(const void* Tval, char* &dest) const;
virtual void putPacked_vec(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                           char* &dest) const;

void get_inline(const char* &src, void* Tval) const;
void getPacked_inline(const char* &src, void* Tval) const;
```

```
void get_vec_inline(const char* &src, void* Tvec, APT_UInt32 veclen,
                    APT_UInt32 sizeofT) const;
void get_vec_inline_input(const char* &src, void* Tvec, APT_UInt32 veclen,
                          APT_UInt32 sizeofT) const;
void getPacked_vec_inline(const char* &src, void* Tvec, APT_UInt32 veclen,
                          APT_UInt32 sizeofT) const;
APT_UInt32 bufLength_vec_inline(const void* Tvec, APT_UInt32 veclen,
                            APT_UInt32 sizeofT) const;
APT_UInt32 bufLengthPacked_vec_inline(const void* Tvec, APT_UInt32 veclen,
                                  APT_UInt32 sizeofT) const;
void put_vec_inline(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                    char* &dest) const;
void putPacked_vec_inline(const void* Tvec, APT_UInt32 veclen, APT_UInt32 sizeofT,
                          char* &dest) const;

// little utilities
static int alignPad(int offset, int mask)
{
  return (-offset) & mask;    // exploits 2s complement negation
}
/* given an offset and a mask, returns the amount of padding, that
   if added to the offset, yields a new offset that is aligned
   according to the given mask:

   mask=0: any alignment
   mask=1: word alignment (even offset)
   mask=3: quad alignment (offset divisible by 4)
   mask=7: oct alignment  (offset divisible by 8)
   etc.

   For example, if mask=3 (quad alignment), then this function computes:

     offset   pad (return value)      offset+pad
     ------   ------------------      ----------
       0              0                   0
       1              3                   4
       2              2                   4
       3              1                   4
       4              0                   4
       5              3                   8
       6              2                   8
      etc            etc                 etc

   Try it for yourself!
*/

/* The length of a prefixed field is stored using either 1 byte or 4
   bytes.  If the size is less than 128, it is stored in 1 byte,
   whose uppermost bit will be clear.

   If the size is 128 or greater, it is stored in 4 bytes.  The
   uppermost bit of the first stored byte will be set, as a flag
   indicating that the size is stored in 4 bytes.  Therefore, sizes
   up to 0x7fffffff (2 gig) can be accommodated.
```

```
     Sizes are always stored in big-endian form, regardless of what kind
     of processor we're running on.
  */
  static APT_UInt32 readSize(const char* &src)
  { if (*src & 0x80)
    { // 4-byte size
      APT_UInt32 answer;
      char* dest = (char*) &answer;
#if defined (LITTLEENDIAN)
      dest[3] = src[0] & 0x7f;
      dest[2] = src[1]; dest[1] = src[2]; dest[0] = src[3];
#elif defined (BIGENDIAN)
      dest[0] = src[0] & 0x7f;
      dest[1] = src[1]; dest[2] = src[2]; dest[3] = src[3];
#else
#error "unknown endian"
#endif
      src += 4;
      return answer;
    }
    else // 1-byte size
      return *src++;
  }
  static void writeSize(char* &dest, APT_UInt32 size)
  { if (size >= 0x80)
    { APT_UInt32 sz = size;
    const char* src = (const char*) &sz;
#if defined (LITTLEENDIAN)
      dest[0] = src[3] | 0x80;
      dest[1] = src[2]; dest[2] = src[1]; dest[3] = src[0];
#elif defined (BIGENDIAN)
      dest[0] = src[0] | 0x80;
      dest[1] = src[1]; dest[2] = src[2]; dest[3] = src[3];
#else
#error "unknown endian"
#endif
      dest += 4;
    }
    else // size < 0x80
      *dest++ = (char) size;
  }

private:
  inline void manageStorage(void* Tval) const;

  int alignMask_;
  APT_UInt32 offsetOfBufferp_;
  APT_UInt32 offsetOfLength_;

  APT_UInt32 offsetOfFlag_;
  void (*delfunc_)(void*);
  APT_UInt8 invertMask_;        // bit set if inversion needed in bit test
  APT_UInt8 bitMask_;           /* isolate bit; zero means not
                                   managing storage */
  APT_DECLARE_RTTI(APT_PrefixedFieldProtocol);
```

```
#ifndef __LINUX__
  APT_DECLARE_NEW_AND_DELETE(APT_PrefixedFieldProtocol);
#endif
};

inline void
APT_PrefixedFieldProtocol::get_inline(const char* &src, void* Tval) const
{ APT_ASSERT_DEBUGONLY(src);
  APT_ASSERT_DEBUGONLY(Tval);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  void** bufferp = (void**) ((char*) Tval + offsetOfBufferp_);
  APT_UInt32* length = (APT_UInt32*) ((char*) Tval + offsetOfLength_);

  APT_UInt32 len = readSize(src);       // this field's content length

  // skip alignment padding, if any (even if size is zero)
  src += alignPad(APT_POINTER_TO_INT32(src), alignMask_);

  if (bitMask_)
    manageStorage(Tval);

  *bufferp = (void*) src;       // this field's content start
  *length = len;
  src += len;                   // advance past this field
}

inline void
APT_PrefixedFieldProtocol::getPacked_inline(const char* &src, void* Tval) const
{ APT_ASSERT_DEBUGONLY(src);
  APT_ASSERT_DEBUGONLY(Tval);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  void** bufferp = (void**) ((char*) Tval + offsetOfBufferp_);
  APT_UInt32* length = (APT_UInt32*) ((char*) Tval + offsetOfLength_);

  APT_UInt32 len = readSize(src);       // this field's content length

  // don't skip alignment padding

  if (bitMask_)
    manageStorage(Tval);

  *bufferp = (void*) src;       // this field's content start
  *length = len;
  src += len;                   // advance past this field
}

inline void
APT_PrefixedFieldProtocol::get_vec_inline(const char* &src, void* Tvec,
                                          APT_UInt32 veclen, APT_UInt32 sizeofT)
const
{ APT_ASSERT_DEBUGONLY(src);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);
```

```
  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    void** bufferp = (void**) ((char*) Tvec + offsetOfBufferp_);
    APT_UInt32* length = (APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    APT_UInt32 len = readSize(src);     // this field's content length

    // skip alignment padding, if any (even if size is zero)
    src += alignPad(APT_POINTER_TO_INT32(src), alignMask_);

    if (bitMask_)
      manageStorage(Tvec);

    *bufferp = (void*) src;     // this field's content start
    *length = len;
    src += len;                 // advance past this field

    Tvec = (char*) Tvec + sizeofT;
  }
}

inline void
APT_PrefixedFieldProtocol::get_vec_inline_input(const char* &src,
                                                void* Tvec,
                                                APT_UInt32 veclen,
                                                APT_UInt32 sizeofT) const
{ APT_ASSERT_DEBUGONLY(src);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    void** bufferp = (void**) ((char*) Tvec + offsetOfBufferp_);
    APT_UInt32* length = (APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    APT_UInt32 len = readSize(src);     // this field's content length

    // skip alignment padding, if any (even if size is zero)
    src += alignPad(APT_POINTER_TO_INT32(src), alignMask_);

    *bufferp = (void*) src;     // this field's content start
    *length = len;
    src += len;                 // advance past this field

    Tvec = (char*) Tvec + sizeofT;
  }
}

inline void
APT_PrefixedFieldProtocol::getPacked_vec_inline(const char* &src, void* Tvec,
                                                APT_UInt32 veclen,
                                                APT_UInt32 sizeofT) const
{ APT_ASSERT_DEBUGONLY(src);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  while (veclen--)
```

```
  { APT_ASSERT_DEBUGONLY(Tvec);
    void** bufferp = (void**) ((char*) Tvec + offsetOfBufferp_);
    APT_UInt32* length = (APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    APT_UInt32 len = readSize(src);     // this field's content length

    // don't skip alignment padding

    if (bitMask_)
      manageStorage(Tvec);

    *bufferp = (void*) src;     // this field's content start
    *length = len;
    src += len;                 // advance past this field

    Tvec = (char*) Tvec + sizeofT;
  }
}

inline void
APT_PrefixedFieldProtocol::manageStorage(void* Tval) const
{
  char* flagByte = (char*) Tval + offsetOfFlag_;
  char fb = *flagByte;

  if ((fb & bitMask_) ^ invertMask_)
  {
    (*delfunc_)(Tval);
    *flagByte = fb ^ bitMask_;  // toggle flag to other state
  }
}

inline APT_UInt32
APT_PrefixedFieldProtocol::bufLength_vec_inline(const void* Tvec,
                                                APT_UInt32 veclen,
                                                APT_UInt32 sizeofT) const
{ APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  APT_UInt32 answer = 0;
  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    const APT_UInt32* length = (const APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    if (*length < 128)
      answer += 1;
    else
      answer += 4;

    // assume worst-case alignment padding
    answer += alignMask_;

    answer += *length;

    Tvec = (char*) Tvec + sizeofT;
  }
```

```
  return answer;
}

inline APT_UInt32
APT_PrefixedFieldProtocol::bufLengthPacked_vec_inline(const void* Tvec,
                                                      APT_UInt32 veclen,
                                                      APT_UInt32 sizeofT) const
{ APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  APT_UInt32 answer = 0;
  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    const APT_UInt32* length = (const APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    if (*length < 128)
      answer += 1;
    else
      answer += 4;

    // don't skip alignment padding

    answer += *length;

    Tvec = (char*) Tvec + sizeofT;
  }

  return answer;
}

inline void
APT_PrefixedFieldProtocol::put_vec_inline(const void* Tvec, APT_UInt32 veclen,
                                          APT_UInt32 sizeofT, char* &dest) const
{ APT_ASSERT_DEBUGONLY(dest);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    const void** bufferp = (const void**)
      ((const char*) Tvec + offsetOfBufferp_);
    const APT_UInt32* length = (const APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    writeSize(dest, *length);

    // skip alignment padding, if any (even if size is zero)
    int pad = alignPad(APT_POINTER_TO_INT32(dest), alignMask_);
    while (pad--) *dest++ = 0;

    if (*length > 0)
    {
      memcpy(dest, *bufferp, *length);
      dest += *length;
    }

    Tvec = (char*) Tvec + sizeofT;
```

```
  }
}

inline void
APT_PrefixedFieldProtocol::putPacked_vec_inline(const void* Tvec,
                                                APT_UInt32 veclen,
                                                APT_UInt32 sizeofT,
                                                char* &dest) const
{ APT_ASSERT_DEBUGONLY(dest);
  APT_ASSERT_DEBUGONLY(offsetOfBufferp_ != offsetOfLength_);

  while (veclen--)
  { APT_ASSERT_DEBUGONLY(Tvec);
    const void** bufferp = (const void**)
      ((const char*) Tvec + offsetOfBufferp_);
    const APT_UInt32* length = (const APT_UInt32*) ((char*) Tvec + offsetOfLength_);

    writeSize(dest, *length);

    // don't skip alignment padding

    if (*length > 0)
    {
      memcpy(dest, *bufferp, *length);
      dest += *length;
    }

    Tvec = (char*) Tvec + sizeofT;
  }
}

#endif // APT_PROTOCOL_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TYPE_TIME_H
#define APT_TYPE_TIME_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_TIME_H
#include <apt_util/time.h>
#endif

class APT_TimeDescriptor : public APT_FieldTypeDescriptor
{
    // Doesn't appear to require destructor as hasMicro_ is the
    // only parameter and it doesn't need to be managed on destruction.

    APT_DECLARE_RTTI(APT_TimeDescriptor);
    APT_DECLARE_PERSISTENT(APT_TimeDescriptor);

  public:

    // See APT_FieldTypeDescriptor in apt_framework/type/descriptor.h
    // for documentation of these.

    APT_TimeDescriptor();

    virtual void *allocValueType(APT_UInt32 veclen) const;
    virtual void freeValueType(void *Tvec, APT_UInt32 veclen) const;

    virtual void copyValueType(const void* Tsrc, void* Tdest) const;
    virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                   APT_UInt32 veclen) const;

    virtual void clearValueType(void* Tdest) const;
    virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;

    virtual void setInBandNull(void *Tdest) const;
    virtual void unparse(ostream&) const;
    virtual bool isBuiltInType() const; // Will be true.
```

```
    static int registerDescriptor();
    // effect   Registers an instance of the integer descriptor on the
    //          first call; subsequent calls are nops.

    virtual APT_FieldProtocol *protocol() const;

    // Check to see if set is required as well?
    bool hasMicrosecondResolution() const { return hasMicro_; }

    void setHasMicrosecondResolution(bool); // default is false

 // protected:

    virtual APT_FieldTypeDescriptor* clone() const;
    virtual void parse_(const char*, APT_ParseError*);
    virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
    virtual APT_UInt32 hash_() const;

    virtual void prepareForInput(void* Tval) const;
    virtual void prepareForOutput(void* Tval) const;

  private:

    bool hasMicro_;
};

// Make sure everyone including this file calls the registration
// function (the registration function only does something on the first
// call).  We could make this only be called once, but only at the risk of
// not having it registered in some files where people need it.
static int APT_sRegisterTime = APT_TimeDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_Time, time, Time);

#endif // APT_TYPE_TIME_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TYPE_TIMESTAMP_H
#define APT_TYPE_TIMESTAMP_H

#ifndef APT_DESCRIPTOR_H
#include <apt_framework/type/descriptor.h>
#endif

#ifndef APT_ACCESSORBASE_H
#include <apt_framework/accessorbase.h>
#endif

#ifndef APT_TIME_H
#include <apt_util/time.h>
#endif

class APT_TimeStampDescriptor : public APT_FieldTypeDescriptor
{
    // Doesn't appear to require destructor as hasMicro_ is the
    // only parameter and it doesn't need to be managed on destruction.

    APT_DECLARE_RTTI(APT_TimeStampDescriptor);
    APT_DECLARE_PERSISTENT(APT_TimeStampDescriptor);

  public:

    // See APT_FieldTypeDescriptor in apt_framework/type/descriptor.h
    // for documentation of these.

    APT_TimeStampDescriptor();

    virtual void *allocValueType(APT_UInt32 veclen) const;
    virtual void freeValueType(void *Tvec, APT_UInt32 veclen) const;

    virtual void copyValueType(const void* Tsrc, void* Tdest) const;
    virtual void copyValueType_vec(const void* Tsrc, void* Tdest,
                                   APT_UInt32 veclen) const;

    virtual void clearValueType(void* Tdest) const;
    virtual void clearValueType_vec(void* Tdest, APT_UInt32 veclen) const;

    virtual void setInBandNull(void *Tdest) const;
    virtual void unparse(ostream&) const;
    virtual bool isBuiltInType() const; // Will be true.

    static int registerDescriptor();
    // effect   Registers an instance of the integer descriptor on the
    //          first call; subsequent calls are nops.

    virtual APT_FieldProtocol *protocol() const;
```

```
    // Check to see if set is required as well?
    bool hasMicrosecondResolution() const { return hasMicro_; }

    void setHasMicrosecondResolution(bool); // default is false

 // protected:

    virtual APT_FieldTypeDescriptor* clone() const;
    virtual void parse_(const char*, APT_ParseError*);
    virtual bool isEqual_(const APT_FieldTypeDescriptor*) const;
    virtual APT_UInt32 hash_() const;

    virtual void prepareForInput(void* Tval) const;
    virtual void prepareForOutput(void* Tval) const;

  private:

    bool hasMicro_;
};

// Make sure everyone including this file calls the registration
// function (the registration function only does something on the first
// call).  We could make this only be called once, but only at the risk of
// not having it registered in some files where people need it.
static int APT_sRegisterTimeStamp = APT_TimeStampDescriptor::registerDescriptor();

APT_DECLARE_ACCESSORS(APT_TimeStamp, timestamp, TimeStamp);

#endif // APT_TYPE_TIME_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.
//
// $Id: fieldlist.h,v 1.10 1998/05/15 16:49:43 smr Exp $

#ifndef APT_FIELDLIST_H
#define APT_FIELDLIST_H

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

class ostream;
class APT_FieldSelector;

//
// APT_FieldList
//

/*
   OVERVIEW

   This utility class converts a string specifying a subset of fields
   in a context to an enumerated list of fields.  It provides iterator
   operations over the resulting list of fields to ease field oriented
   operations like binding accessors.  It is intended for use by
   operators which desire access to a dynamic subset of fields in a
   schema.

   Field lists
   Field lists can be constructed by explicitly specifying each field
   in the list or by using character strings that specified a "field
   list specification". These specifications can contain comma-delimited
   lists of field names, ranges of fields (like "foo - bar") or "*" (meaning
   all fields in the context). The specification is "expanded" in some
   "context", which is either a schema or an already-expanded field list.

   A field specification can consist of the following elements, separated
   by commas:

    - individual field identifiers - simply one or more field
      names; field names are fully qualified paths to scalars or to
      fixed length vectors (ex: address, city, ...).

    - field ranges - two field identifiers separated by a hyphen.  The
```

```
      first field must appear earlier in the context than the second.
      Elements within a field range includes all fields whose ordinal
      position in a context is between the field starting the range and
      the field ending the range, inclusively.  (example: income - age)

    - all fields in a context - denoted by *.

    Subset specification types may be mixed; fields are guaranteed to
    appear only once in the resulting field list.  Detecting
    duplicates results in an error.
*/

class APT_Schema;
class APT_FieldListImpl;

class APT_FieldList : public APT_Persistent
{
    APT_DECLARE_RTTI(APT_FieldList);
    APT_DECLARE_PERSISTENT(APT_FieldList);

 public:

    class Error : public APT_Exception
    {
      APT_DECLARE_EXCEPTION(APT_FieldList::Error);
    public:
      Error();
      Error(const char * errorMessage);
      ~Error();

      bool errorOccurred() const { return errorOccurred_; }

      bool errorOccurred_;      // should be private; don't use!

    protected:
      APT_String description_() const;

    private:
      APT_String message_;
    };


    //
    // Constructors / destructor / assignment operator
    //


    APT_FieldList();
    /*
       effects  Construct an empty field list.
    */

    APT_FieldList(const APT_FieldSelector& field);
    /*
```

```
      effects  Construct a field list containing the single
               specified field.
   */

   APT_FieldList(const char * fieldListSpec, APT_FieldList::Error* errp=0);
   /*
      effects  Construct a field list containing the supplied field
               list specification. The specification is syntax checked
               but otherwise unprocessed.
               If a parse error occurs, an error is thrown unless the
               optional error argument is non-null, in which case the
               error argument is assigned the error that would have
               been thrown, and the error argument's errorOccurred_
               attribute is set to true.
      throws   APT_FieldList::Error (for syntax errors; see syntax rules
                 listed in overview.)
   */

   APT_FieldList(const APT_FieldList & list);
   /*
      effects  Copy constructor. The result is a deep copy; no state is
               shared between the new and original field lists.
   */

   ~APT_FieldList();
   /*
      effects  Destroy a field list.
   */

   APT_FieldList & operator=(const APT_FieldList & list);
   /*
      effects  Assign one field list to another. The result is a deep copy.
   */



   //
   // Accessors
   //

   bool isExpanded() const;
   /*
      effects  Returns false if the field list was constructed from a
               field list specification (i.e., a character string) and
               expand() has not been called. Otherwise returns true.
               The implementation may choose to return true if a
               field list was constructed from a character string that
               contained no wildcards or ranges.
   */

   APT_Int32 numFields() const;
   /*
      effects  Returns the number of fields in the list.

      requires isExpanded() == true
```

```
   */

   APT_FieldSelector field(APT_Int32 fieldIndex) const;
   /*
      effects  Returns the field selector for the field having the specified
               index.

      requires isExpanded() == true

               0 <= fieldIndex < numFields()
   */

   APT_Int32 fieldIndex(const APT_FieldSelector & field) const;
   /*
      effects  If the specified field is in the list, the field's
               index is returned. Otherwise, -1 is returned.

               If "F" is a field selector for a field in list "L",
               then this is true:

               F = L.field(L.fieldIndex(F));

      requires isExpanded() == true
   */



   //
   // Mutators
   //

   void expand(const APT_Schema & schema, APT_FieldList::Error* errp=0);
   void expand(const APT_FieldList & list, APT_FieldList::Error* errp=0);
   /*
      effects  Expands the field list in the specified context.
               Causes subsequent calls to isExpanded() to be true.
               If isExpanded() is already true, this is a no-op.
               If an error is detected (see below), an error is
               thrown unless the optional error argument is non-null,
               in which case the error argument is assigned the error
               that would have been thrown, and the error argument's
               errorOccurred_ attribute is set to true.

      throws   APT_FieldList::Error: One or more of the following
               problems has been detected:
               - One or more ranges in the field list was not valid
                 with respect to the specifed context.
               - The expanded list contained duplicate fields.
               - In the second form, the field list supplied as the context
                 was not expanded (its isExpanded() == false).

      note     If an error occurs in the processing of this operation,
                     the APT_FieldList is left in an unspecified state.  Programs
               that wish to recover from failure should use a copy of the
               APT_FieldList.
```

```
   */


   void addFields(const APT_FieldList & list, APT_FieldList::Error* errp=0);
   /*
      effects  Appends the specified field list to this field list.

               The resulting list will have isExpanded() true if
               both this list and the argument list have isExpanded()
               true; otherwise, isExpanded() will be false.

               If isExpanded() will be true for the resulting list,
               the list is checked for duplicate fields. Otherwise
               this check is deferred until expand() is called.

      throws   APT_FieldList::Error (for duplicate fields)

      note     If an error occurs in the processing of this operation,
                      the APT_FieldList is left in an unspecified state.  Programs
               that wish to recover from failure should use a copy of the
               APT_FieldList.
   */


   void keepFields(const APT_FieldList & list, APT_FieldList::Error* errp=0);
   /*
      effects  The incoming list is expanded if necessary using the
               current list as the context. The resulting list is
               checked to be sure all of its fields are in the current
               list. If so, the resulting list replaces the current list;
               otherwise an error is thrown.

      requires this->isExpanded() == true

      throws   APT_FieldList::Error (incoming list contains fields not
               in the current list)

      note     If an error occurs in the processing of this operation,
                      the APT_FieldList is left in an unspecified state.  Programs
               that wish to recover from failure should use a copy of the
               APT_FieldList.
   */


   void dropFields(const APT_FieldList & list, APT_FieldList::Error* errp=0);
   /*
      effects  The incoming list is expanded if necessary using the
               current list as the context. The resulting list is
               checked to be sure all of its fields are in the current
               list. If so, all fields in the resulting list are dropped
               from the current list; otherwise an error is thrown.

      requires this->isExpanded() == true
```

```
      throws   APT_FieldList::Error (incoming list contains fields not
               in the current list)

      note     If an error occurs in the processing of this operation,
                      the APT_FieldList is left in an unspecified state.  Programs
               that wish to recover from failure should use a copy of the
               APT_FieldList.
   */


   void dropFields(const APT_FieldList & list,
                   APT_FieldList & missingFields,
                   APT_FieldList::Error* errp=0);
   /*
      effects  The incoming list is expanded if necessary using the
               current list as the context. The resulting list is
               checked to be sure all of its fields are in the current
               list. If so, all fields in the resulting list are dropped
               from the current list; otherwise an error is thrown.

               This differs from the previous version of dropFields in
               that any fields in the drop list that are not in the
               current list will be appended to the "missingFields"
               list; since this condition is harmless, an error will
               not be thrown.

      requires this->isExpanded() == true
               missingFields.numFields() == 0

      throws   APT_FieldList::Error (incoming list contains fields not
               in the current list)

      note     If an error occurs in the processing of this operation,
                      the APT_FieldList is left in an unspecified state.  Programs
               that wish to recover from failure should use a copy of the
               APT_FieldList.
   */


   void print(ostream &);

   APT_String unparse() const;
   /*
      returns  A string representing the fieldlist that can be passed
                      to the constructor.
    */

   friend ostream &operator<< (ostream &s, const APT_FieldList &l);

 private:

   APT_FieldListImpl * impl_;
};
```

```
#endif                          // APT_FIELDLIST_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ARCHIVE_H
#define APT_ARCHIVE_H

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ASSERT_H
#include <apt_util/assert.h>
#endif

#ifndef APT_EXCEPTION_H
#include <apt_util/exception.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#include <fstream.h>
#include <sys/time.h>

class APT_Persistent;
class APT_ArchiveRep;

class APT_Archive
/* Stream class for object persistence, providing serialization
   operators for the built-in data types.  The serialization operators
   efficiently and portably convert data to and from a serial byte
   sequence.

   A single APT_Archive class is used for both input (loading) and
   output (storing).  A runtime attribute determines the mode of an
   archive object; the mode is established when the archive object is
   created and cannot be changed.

   For each overloaded data type, a single serialization operator can
   be used for both storing and loading.  The archive's mode
   determines in which direction data is actually flowing.  For
   example, one of the operators provided by the APT_Archive public
   interface is:

     APT_Archive& operator|| (APT_Archive& ar, float& dat);
```

Such operators can be composed and used as follows:

```
  class Complex
  {
    float re_, im_;
  public:
    ...
    friend APT_Archive& operator|| (APT_Archive& ar, Complex& c)
      { return ar || c.re_ || c.im_; }
    ...
  };

  APT_FileArchive ar1("input.dat", APT_Archive::eLoading);
  APT_FileArchive ar2("output.dat", APT_Archive::eStoring);
  Complex c;
  ar1 || c;                 // load c from input.dat
  ar2 || c;                 // store c to output.dat
```

Key point: to make the Complex class persistent, it is only necessary to define one serialization operator. Most other persistence packages would require at least two nearly identical functions to be written: one for loading, the other for storing. With APT_Archive, programmer workload (and the possibility of error) is reduced, since only one serialization operator need be implemented.

In addition to the bidirectional operator||() overloads, it is customary to also provide operator<<() and operator>>() functions implemented in terms of operator||(). The above example would be written:

```
  class Complex
  {
    float re_, im_;
  public:
    ...
    friend APT_Archive& operator|| (APT_Archive& ar, Complex& c)
      { return ar || c.re_ || c.im_; }
    ...
  };
  APT_DIRECTIONAL_SERIALIZATION(Complex);

  APT_FileArchive ar1("input.dat", APT_Archive::eLoading);
  APT_FileArchive ar2("output.dat", APT_Archive::eStoring);
  Complex c;
  ar1 >> c;                 // load c from input.dat
  ar2 << c;                 // store c to output.dat
```

The APT_DIRECTIONAL_SERIALIZATION macro adds operator>>() and operator<<() as inline synonyms for the more fundamental operator||(). The class developer still only needs to think about operator||(). The << and >> synonyms come for free with the macro.

```
   When writing operator|| overloads, it is customary to include the
   following boilerplate code:

     APT_Archive& operator|| (APT_Archive& ar, Complex& c)
     {
       if (ar.isEof()) APT_THROW( APT_Archive::Eof(ar.ident()) );

       APT_UInt8 ver = 0;
       ar || ver;
       if (ver != 0) APT_THROW(APT_Archive::BadVersion(ar.ident()));

       return ar || c.re_ || c.im_;
     }

   The EOF test is a useful sanity check.  Serializing a version byte
   allows future versions of the Complex class to evolve its
   serialized representation while being able to load earlier
   serialized formats.

   In addition to allowing built-in types to be serialized, and simple
   objects to be serialized via class-specific overloads of
   operator||(), APT_Archive collaborates with the APT_Persistent base
   class, yielding a full object persistence system.  See persist.h
   and persistence.html for more information.
*/
{
public:
  enum Mode { eLoading, eStoring };

protected:
  APT_Archive(Mode);
  /*
     effect    Initializes this archive for the indicated mode of
               operation.
               Until bindStream() is called, this archive isNull() and
               is essentially unusable.
  */

public:
  virtual ~APT_Archive();
  /*
     note      Does not attempt to flush(), because derived dtor has
               already killed our stream.
  */

  void flush();
  /*
     effect    Flushes any pending output.  No effect if loading.
  */

  Mode mode() const { return storing_ ? eStoring : eLoading; }
```

```
bool isLoading() const { return !storing_; }
bool isStoring() const { return storing_; }
/*
   effect    Tells the direction of this archive.
*/

void checkStoring() const { if (!storing_) checkStoring_(); }
void checkLoading() const { if (storing_) checkLoading_(); }
/*
   effect    Enforces (via a user requirement check) that the archive
             is in the respective mode.
*/

virtual APT_String ident() const;
/*
   effect    Returns a string identifying this archive.
             Base class implementation returns "Loading archive" or
             "Storing archive"; derived classes can add additional
             information.
*/

bool isNull() const { return !sbuf_; }
/*
   effect    Returns true if bindStream() was not called, if
             unbind() was called, or a loading archive's magic string
             was invalid.
             A null APT_Archive is essentially unusable.
*/

bool isMagicBad() const { return badMagic_; }
/*
   effect    Returns true if this is a loading archive constructed
             with a magicString not matching the stored magic string.
*/

bool isEof() const { return !storing_ && sbuf_->atEof(); }
/*
   effect    Tells if this archive has reached EOF.  Will never be
             true for storing archives.
*/

bool isFull() const { return full_; }
/*
   effect    Tells if this archive can no longer be written to
             (typically because of a full disk).  Will never be true
             for loading archives.
*/

bool failOnBadClass() const;
void setFailOnBadClass(bool);
/*
   effect    When loading an APT_Persistent pointer, if the object's
```

```
              class (as recorded in the archive stream) does not
              exist in the loading program, a "bad class" error is
              said to have occurred.
              If failOnBadClass() is true (default), then the
              APT_Archive::BadClass exception is thrown when such a
              pointer is loaded.
              If failOnBadClass() is false, then the loaded pointer
              is set to null and the object's serialized
              representation is skipped (allowing loading to
              continue).
  */


  /*
    Serialization operators for fundamental types.  Storing always
    uses native byte ordering for integers and IEEE format (4 and 8
    bytes) for reals; loading converts archive's integer byte ordering
    (as indicated in the archive header) and IEEE reals to native
    form.

    Consequently, storing operators are very simple and are mostly
    inline; loading functions are potentially more complex and are
    generally not inline.

    When loading, if EOF occurs before all bytes of the the data are
    loaded, then data value loaded is unspecified, and the archive's
    isEof() becomes true.

    Requirements for all serialization operations:
      isNull() must be false
      When loading, the archive must be positioned to where the
        corresponding store operation was performed.
      When loading, isEof() must be false
  */

  // operators to serialize one value

  friend APT_Archive& operator|| (APT_Archive& ar, APT_UInt8& d);
  friend APT_Archive& operator|| (APT_Archive& ar, APT_Int8& d);
  friend APT_Archive& operator|| (APT_Archive& ar, char& d);

  friend APT_Archive& operator|| (APT_Archive& ar, APT_UInt16& d);
  friend APT_Archive& operator|| (APT_Archive& ar, APT_Int16& d);

  friend APT_Archive& operator|| (APT_Archive& ar, APT_UInt32& d);
  friend APT_Archive& operator|| (APT_Archive& ar, APT_Int32& d);

  friend APT_Archive& operator|| (APT_Archive& ar, APT_UInt64& d);
  friend APT_Archive& operator|| (APT_Archive& ar, APT_Int64& d);

#if defined(APT_SERIALIZE_TIME_T)
  friend APT_Archive& operator|| (APT_Archive& ar, time_t& d);
```

```
#endif

#ifdef APT_INT32_IS_NOT_INT
  friend APT_Archive& operator|| (APT_Archive& ar, unsigned int& d);
  friend APT_Archive& operator|| (APT_Archive& ar, int& d);
#endif

#ifdef __BOOL__                 // bool is genuine (not a typedef)
  /* treat bool same as int (so that storage format is the same as
     when bool is just a typedef for int) */
  friend APT_Archive& operator|| (APT_Archive& ar, bool& d);
#endif

  friend APT_Archive& operator|| (APT_Archive& ar, float& d);
  friend APT_Archive& operator|| (APT_Archive& ar, double& d);

  // operators to serialize an array of values

  // TBD: need const array overloads?
  void xfer(APT_UInt8* arr, APT_UInt32 elts);
  void xfer(APT_Int8* arr, APT_UInt32 elts) { xfer((APT_UInt8*) arr, elts); }
  void xfer(char* arr, APT_UInt32 elts) { xfer((APT_UInt8*) arr, elts); }

  void xfer(APT_UInt16* arr, APT_UInt32 elts);
  void xfer(APT_Int16* arr, APT_UInt32 elts) { xfer((APT_UInt16*) arr, elts); }

  void xfer(APT_UInt32* arr, APT_UInt32 elts);
  void xfer(APT_Int32* arr, APT_UInt32 elts) { xfer((APT_UInt32*) arr, elts); }
  void xfer(APT_UInt64* arr, APT_UInt32 elts);
  void xfer(APT_Int64* arr, APT_UInt32 elts) { xfer((APT_UInt64*) arr, elts); }

/* Unneeded because int always == int32
  void xfer(unsigned int* arr, APT_UInt32 elts);
  void xfer(int* arr, APT_UInt32 elts) { xfer((unsigned int*) arr, elts); }
*/

  void xfer(float* arr, APT_UInt32 elts);
  void xfer(double* arr, APT_UInt32 elts);

  void deleteWhenDoneSerializing(APT_Persistent*);
  /*
     effect    Adds the given object to a set of objects to be
               deleted when this archive is told to flush its
               serialized object memoizations (see persist.h).
  */

  void repInvariant() const;

  class ErrorBase : public APT_Exception
  { APT_DECLARE_RTTI(APT_Archive::ErrorBase);
    APT_DECLARE_EXCEPTION(APT_Archive::ErrorBase);
  public:
```

```
  ~ErrorBase();
protected:
  ErrorBase(const char* ident);
  virtual APT_String description_() const;
  APT_String ident_;
};

class BadData : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::BadData);
  APT_DECLARE_EXCEPTION(APT_Archive::BadData);
public:
  BadData(const char* ident);
  ~BadData();
};

class BadVersion : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::BadVersion);
  APT_DECLARE_EXCEPTION(APT_Archive::BadVersion);
public:
  BadVersion(const char* ident);
  ~BadVersion();
};

class BadPointer : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::BadPointer);
  APT_DECLARE_EXCEPTION(APT_Archive::BadPointer);
public:
  BadPointer(const char* expected, const char* actual,
             const char* ident);
  ~BadPointer();
protected:
  virtual APT_String description_() const;
private:
  APT_String expected_;
  APT_String actual_;
};

class Eof : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::Eof);
  APT_DECLARE_EXCEPTION(APT_Archive::Eof);
public:
  Eof(const char* ident);
  ~Eof();
};

class Full : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::Full);
  APT_DECLARE_EXCEPTION(APT_Archive::Full);
public:
  Full(const char* ident);
  ~Full();
};
```

```
class BadClass : public ErrorBase
{ APT_DECLARE_RTTI(APT_Archive::BadClass);
  APT_DECLARE_EXCEPTION(APT_Archive::BadClass);
public:
  BadClass(const char* classname, const char* ident);
  ~BadClass();
protected:
  virtual APT_String description_() const;
private:
  APT_String classname_;
};


// stream class for transorting archived bytes
class Stream
{
public:
  virtual ~Stream();

  // operations for input stream

  const char* obtainInput(APT_Int32 request, APT_Int32 *actual,
                          APT_String* errReason=0)
  { if (request <= remaining_)
    { if (actual) *actual = request;
      return client_;
    }
    else return obtainInput_(request, actual, errReason);
  }
  void consumedInput(APT_Int32 amount)
  { APT_ASSERT_DEBUGONLY(amount >= 0);
    APT_ASSERT_DEBUGONLY(amount <= remaining_);
    client_ += amount;
    remaining_ -= amount;
  }

  const char* obtainAndConsume(APT_Int32 request)
  { APT_Int32 actual;
    const char* answer = obtainInput(request, &actual);
    if (request == actual)
    {
      consumedInput(request);
      return answer;
    }
    else
    {
      consumedInput(actual);
      return 0;
    }
  }
```

```
  bool atEof() const;

  virtual bool isSeekable() const; /* true in base class (memory Stream);
                                      derivations should override as
                                      appropriate */
  APT_Int64 seekPos() const;  // requires: isSeekable() true
  void seek(APT_Int64);       /* requires: isSeekable() true; may not
                                 seek outside of current stream extent */

protected:
  virtual APT_Int32 read(char* buf, APT_Int32 request,
                         APT_String* errReason);

  // must override if derived isSeekable() can be true
  virtual APT_Int64 seekPos_() const;
  virtual void seek_(APT_Int64);

private:
  const char* obtainInput_(APT_Int32 request, APT_Int32 *actual,
                           APT_String* errReason);


public:
  // operations for output stream

  char* prepareOutputBuffer(APT_Int32 request,
                            APT_String* errReason=0)
  {
    if (request <= remaining_) return client_;
    else return prepareOutputBuffer_(request, errReason);
  }
  void wroteOutput(APT_Int32 amount)
  {
    APT_ASSERT_DEBUGONLY(amount >= 0);
    APT_ASSERT_DEBUGONLY(amount <= remaining_);
    client_ += amount;
    remaining_ -= amount;
  }
  APT_Status flushOutput(APT_String* errReason=0);

  char* buffer(APT_Int32* length); /* for output memory stream only;
                                      buffer ownership passes to client;
                                      cannot write to this stream after
                                      calling buffer() */
protected:
  virtual APT_Int32 write(const char* buf, APT_Int32 request,
                          APT_String* errReason);
  // override should also flush

private:
  char* prepareOutputBuffer_(APT_Int32 request,
                             APT_String* errReason);
```

```
  public:
    enum Direction { eInput, eOutput };
  protected:
    Stream(Direction);          // use virtual functions for read/write

  public:
    Stream(char* buf, APT_Int32 len); /* input from supplied buffer;
                                         Stream takes ownership */
    Stream(const char* buf, APT_Int32 len); /* input from supplied buffer;
                                               client retains ownership */
    Stream();                   // output to Stream-managed buffer

  private:
    // prohibit copy/assign
    Stream(const Stream&);
    Stream& operator= (const Stream&);

    void resizeBuffer(APT_Int32 newSize);

    Direction dir_;
    bool memoryStream_;         // special base implementation
    bool readEof_;
    bool ownBuf_;
    char* buf_;
    char* client_;
    APT_Int32 remaining_;
    APT_Int32 bufSize_;
  };

protected:
  void bindStream(Stream* buf, const char* magicString, APT_UInt32 magicLength);
  /*
     effect    Initializes this archive to use the indicated stream.
               The client retains ownership of buf.
               If buf is null, then this archive isNull() and no
               serializations may be perfomed on it.
               When storing, the magicString is stored at the
               beginning of the archive.
               When loading, the stored magic string is compared to
               the constructor argument; if they do not match,
               isNull() and isMagicBad() will be true.
     throws    BadData: Bad archive header detected on loading
                 archive.
               BadVersion: Bad archive version detected on loading
                 archive.
               Eof: Eof occurred while reading archive header.
     requires  buf non-null
               isNull() must be true.
               buf must be set up for reading (if storing==0) or
               writing (if storing!=0).
               If loading, buf must be positioned to the point where
```

```
              the storing archive was constructed.  Typically this is
              the beginning of the stream.
  */

  void unbind();
  /*
     effect    Causes the sbuf_ pointer to be null.  No serializations
               may be performed after this function is called.
  */

private:
  friend class APT_Persistent;
  friend class APT_ArchiveRep;
  friend class APT_ArchiveUT;   // back door for unit-test

  // prohibit copying
  APT_Archive(const APT_Archive&);
  APT_Archive& operator= (const APT_Archive&);

  void load16(APT_UInt16&);
  void load32(APT_UInt32&);
  void load64(APT_UInt64&);
  void load16(APT_UInt16*, APT_UInt32);
  void load32(APT_UInt32*, APT_UInt32);
  void load64(APT_UInt64*, APT_UInt32);

  void discard(APT_UInt32);

  void recordRefSerialization(APT_Persistent&);

  void forgetSerializations();
  /*
     effect    Causes this archive to discard its memory of
               complex-persistent objects that have been stored
               via pointer on this archive.
  */

  void storePointer(APT_Persistent*);
  /* handles null pointer and already-stored object; otherwise calls
     reallyStorePointer() */

  APT_Persistent* loadPointer();
  /* throws: BadData, Eof, BadClass; tries to skip object in stream */

  static void persistentDtor(APT_Persistent*);
  // tells all archives that the indicated object is going away

  void checkStoring_() const;
  void checkLoading_() const;

public:
  enum IntegerFormat { eBig=0, eLittle=1, ePdp11=2 };
```

```
  enum RealFormat { eIEEE=0 };

private:
  bool storing_;
  bool badMagic_;
  bool full_;
  Stream* sbuf_;
  APT_ArchiveRep* rep_;

  /* data format information loaded from archive header; not used when
     storing */
  IntegerFormat storedInts_;    // byte ordering also affects floats/doubles
  bool nativeOrder_;            /* values stored in byte ordering that
                                   is native to us */
  bool nativeReals_;            // ditto for reals

  int nest_;                    /* nesting of complex-persistent
                                   object serializations; manipulated
                                   directly by APT_Persistent */
  bool failOnBadClass_;

  /* native data format of integers and reals on this processor */
  static IntegerFormat sIntFormat;
  static RealFormat sRealFormat;
  static bool sGotFormats;
};

#define APT_DIRECTIONAL_SERIALIZATION(T)                          \
inline APT_Archive& operator<< (APT_Archive& ar, const T& d)    \
  { ar.checkStoring(); return ar || (T&) d; }           \
inline APT_Archive& operator>> (APT_Archive& ar, T& d)          \
  { ar.checkLoading(); return ar || d; }                \
class APT_dummy_class_decl_so_semicolon_can_be_used_with_macro

APT_DIRECTIONAL_SERIALIZATION(APT_UInt8);
APT_DIRECTIONAL_SERIALIZATION(APT_Int8);
APT_DIRECTIONAL_SERIALIZATION(char);
APT_DIRECTIONAL_SERIALIZATION(APT_UInt16);
APT_DIRECTIONAL_SERIALIZATION(APT_Int16);
APT_DIRECTIONAL_SERIALIZATION(APT_UInt32);
APT_DIRECTIONAL_SERIALIZATION(APT_Int32);
APT_DIRECTIONAL_SERIALIZATION(APT_UInt64);
APT_DIRECTIONAL_SERIALIZATION(APT_Int64);
#ifdef APT_INT32_IS_NOT_INT
APT_DIRECTIONAL_SERIALIZATION(int);
APT_DIRECTIONAL_SERIALIZATION(unsigned int);
#endif
APT_DIRECTIONAL_SERIALIZATION(float);
APT_DIRECTIONAL_SERIALIZATION(double);
#ifdef __BOOL__                 // bool is genuine (not a typedef)
APT_DIRECTIONAL_SERIALIZATION(bool);
#endif
```

```
/**** concrete derivations ****/

class APT_FileArchive : public APT_Archive
{
public:
  APT_FileArchive(const char* name, APT_Archive::Mode,
                  const char* magicString=0, APT_UInt32 magicLength=0);
  /*
     effect    Opens (loading) or creates (storing) the indicated
               file.  The file is opened in binary mode.
               If storing, the file is created, truncating an
               existing file (if any) to zero-length.
               When storing, the magicString is stored at the
               beginning of the archive.
               When loading, the stored magic string is compared to
               the constructor argument; if they do not match,
               isNull() and isMagicBad() will be true.
     errors    If there is any problem opening or creating (for
               loading and storing, respectively) the indicated file,
               this archive is created in the null (unusable) state.
     throws    BadData: Bad archive header detected on loading
                 archive.
               BadVersion: Bad archive version detected on loading
                 archive.
               Eof: Eof occurred while reading archive header.
     requires  name non-null
  */

  APT_FileArchive(ofstream& ofile, APT_Archive::Mode m,
                  const char* magicString=0, APT_UInt32 magicLength=0);
  /*
     effect    Binds this archive to the indicated output file, which
               must be open for writing.
               The client retains control over the ofile object's
               lifetime.
               The magicString is stored at the beginning of the
               archive.
     requires  m must be eStoring.
               For the duration of this archive's lifetime, the client
               must not tamper with the ofile object's contents or
               seek position.
  */
  APT_FileArchive(ifstream& ifile, APT_Archive::Mode m,
                  const char* magicString=0, APT_UInt32 magicLength=0);
  /*
     effect    Binds this archive to the indicated input file, which
               must be open for reading.
               The client retains control over the ifile object's
               lifetime.
               The stored magic string is compared to the constructor
```

```
              argument; if they do not match, isNull() and
              isMagicBad() will be true.
   throws     BadData: Bad archive header detected on loading
                archive.
              BadVersion: Bad archive version detected on loading
                archive.
              Eof: Eof occurred while reading archive header.
   requires   m must be eLoading.
              ifile must be seeked to a point at which a storing
              APT_FileArchive was created.
              For the duration of this archive's lifetime, the client
              must not tamper with the ifile object's contents or
              seek position.
*/

~APT_FileArchive();
/*
   note       For an output file archive, flushes the content to the
              destination via fsync(3).
*/

virtual APT_String ident() const; // override
/*
   effect     Returns a string identifying this archive's filename
              (if any) and load/store direction.
*/

APT_String filename() const;
void setFilename(const char*);
/*
   effect     Accesses the filename used to construct this file
              archive.
              The get function returns an empty string if this file
              archive was constructed via ofstream or ifstream and
              setFilenaem() was not called.
*/

// copy/assign prohibited by base class


class FileStream : public APT_Archive::Stream
{
public:
  FileStream(APT_Archive::Stream::Direction,
              int fd);
  FileStream(APT_Archive::Stream::Direction,
              filebuf*);       // does not take ownership

  virtual bool isSeekable() const;

protected:
  virtual APT_Int32 read(char* buf, APT_Int32 request,
```

```
                          APT_String* errReason);
    virtual APT_Int32 write(const char* buf, APT_Int32 request,
                            APT_String* errReason);
    virtual APT_Int64 seekPos_() const;
    virtual void seek_(APT_Int64);

    int fd_;                    // set if first ctor form is used
    filebuf* fbuf_;             // non-0 iff second ctor form is used

    bool seekable_;             // true for favorable fd_; false for fbuf_
    APT_Int64 initialPos_;
  };

private:
  APT_String name_;
  FileStream* fStr_;
  filebuf* ownedBuf_;
};

class APT_MemoryArchive : public APT_Archive
{
public:
  APT_MemoryArchive(APT_UInt32 initialBufSize=512,
                    const char* magicString=0, APT_UInt32 magicLength=0);
  /*
     effect    Creates a storing memory archive.  As objects are
               stored to this archive, a memory buffer is allocated
               (and reallocated if necessary) to accommodate the
               serialized representation of the objects.  See the
               buffer() function.
               To reduce the number of buffer reallocations, an
               initialBufSize may be supplied.
               The magicString is stored at the beginning of the
               archive.
     requires  initialBufSize > 0
  */

  APT_MemoryArchive(const char* buf, APT_UInt32 length,
                    const char* magicString=0, APT_UInt32 magicLength=0);
  /*
     effect    Creates a loading memory archive.  This archive loads
               from the indicated buffer, signalling isEof() when the
               end is reached.
               The client retains ownership of the buffer.
               The stored magic string is compared to the constructor
               argument; if they do not match, isNull() and
               isMagicBad() will be true.
     throws    BadData: Bad archive header detected on loading
                 archive.
               BadVersion: Bad archive version detected on loading
                 archive.
               Eof: Eof occurred while reading archive header.
```

```
    requires  buf non-null
              length > 0
  */

  APT_MemoryArchive(APT_MemoryArchive* rhs,
                    const char* magicString=0, APT_UInt32 magicLength=0);
  /*
    modifies  rhs.isNull() becomes true
    effect    Constructs a loading memory archive from the argument.
              The stored magic string is compared to the constructor
              argument; if they do not match, isNull() and
              isMagicBad() will be true.
    note      This is supplied for the convenience of folks who are
              testing a class's persistence code.
    requires  rhs non-null
              rhs->isStoring() must be true.
              rhs->isNull() must be false.
  */

  ~APT_MemoryArchive();

  // copy, assignment prohibited by base class

  virtual APT_String ident() const; // override
  /*
    effect    Returns a string identifying this archive as a loading
              or storing memory archive.
  */

  char* buffer(APT_UInt32* length);
  /*
    modifies  length gets buffer's length
    effect    Returns the buffer created by this storing archive.
              This archive becomes null, and the client is
              responsible for delete[]ing the returned buffer.
    requires  isStoring() must be true.
              isNull() must be false.
              length non-null
  */

private:
  Stream* mStr_;
};

// TBD: archive derivation for storing/loading objects over a
// communication port

#endif // APT_ARCHIVE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1998 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ARGVCHECK_H
#define APT_ARGVCHECK_H

/* Standardized argc/argv argument list processor, for use in
   Orchestrate operators' (or partitioners' and collectors')
   processing logic for dealing with their command line arguments.

   The description of an operator's arguments is presented with the
   operator's C++ code via the APT_DEFINE_OSH_NAME macro (see
   apt_framework/osh_name.h).  This information can be reformatted and
   copied into the Server Database by a utility program; someday the
   argument description may live in the Server Database rather than
   being presented in the C++ code and being copied to the Server
   Database.

   A description of the expected arguments is used by the argument
   list processor to guide its interpretation (and error checking) of
   a supplied argument list.  The result is a data structure encoding
   the validated arguments, and possibly a stream of error messages.
   The client traverses the resulting data structure knowing that a
   large set of error checks have already been performed, and that the
   result conforms to the description of what arguments are expected.

   A raw argument vector consists of a sequence of command-line tokens
   that have been tokenized according to the OSH rules (which are an
   extension of Unix shell tokenization rules).  The argument list
   processor attempts to interpret a raw argument vector according to
   the description with which the argument list processor has been
   initialized.

   The argument list processor treats its input argument vector as a
   list of argument items (argitems).  An argitem consists of the
   following components:

     - name.  An argitem's name uniquely identifies what kind of
       argument it is.  The argitem name is usually related to the OSH
       argument tag in a straightforward manner.  For example, the
       tsort operetor's argitem named key is written as -key in OSH.

     - value(s).  An argitem can have some number of values, typically
       zero or one.  Each value is of a given type, with possibly
       constrained values (more on argitem value types and constraints
       below).

       In tsort, the -key argitem takes a field name value, and an
       optional (deprecated) field type value: -key firstname string[10]

     - sub-args.  An argitem can have sub-arguments, in the form of a
       list of argitems all associated with the parent argitem.

       In tsort, the -key argitem has several sub-args, for example:
```

```
    tsort -key firstname -ci -sorted
```

When the argument list processor is used, it produces a data structure encoding the argitems found in its analysis. This data structure takes the form of a property list, where each top-level property's name is that of the argItem it encodes, and whose value is a property list containing more information about that argument (if any).

The overall form of the argument list processor's result is:

```
  { argName1={...}, argName2={...}, ...}
```

Each argitem is represented by a set of properties as follows:

```
  argName1={value=V,          // optional; 0 or more
            subArgs={ argItem={...}, ...} // optional
           }
```

The argument name is always present, as the name of corresponding property in the result list. Value property(s) will be present according to whether this argitem has any value(s); subargs will be present according to whether this argitem has any sub-args. If an argitem does not have a value or sub-args, then it just appears as an empty property in the result list.

For example, consider the tsort command line:

```
  tsort -key foo -ci -sorted -hash -key bar int32 -memory 32 -stats
```

Given a suitable description for tsort's expected command line arguments (see below), the following resulting property list will be generated by the argument list processor:

```
  { key={value=foo, subArgs={ ci, sorted, hash } },
    key={value=bar, value=int32},
    memory={value=32},
    stats
  }
```

The result list presents argitems in the order in which they were encountered in the raw argument vector.

Note that there *must* be a meaningful description of tsort's expected arguments, because otherwise the argument list processor would have no way of knowing that the first -key argitem ends with -hash, and that the second -key argitem ends with int32.

The key to the argument list processor's usefulness, and the limitations of its power, is how one describes to the argument list processor what arguments to expect.

A property list (rather than defining yet another little language) is used to encode a description of how an argument list processor is to interpret raw argument vectors. The property list has the

following form:

```
  { argName={description=STRING,                // required
             value={<value description>},       // optional; 0 or more
             subArgDesc={ argName={...}, ... }, // optional
             minOccurrences=INT,                // optional; default=0
             maxOccurrences=INT,                // optional; default=inf.
             optional,                          // sugar for min/max=0/1
             oshName=STRING,                    // optional
             oshAlias=STRING,                   // optional; 0 or more
             silentOshAlias=STRING,             // optional; 0 or more
             default,                           // optional
             hidden,                            // optional
             deprecated[=IDENT]                 // optional
            },
    argName={...},          // 0 or more argument descriptions
    ...
    otherInfo={             // optional
               exclusive={name, name, ...},  // optional; 0 or more
               exclusive_required={name, name, ...}, // optional; 0 or more
               implies={name, name},         // optional; 0 or more
               description=STRING            // optional
              }
  }
```

Each argument entry has a name (an identifier). Within an argument list, all argument names must be unique. Argument names are matched in a case-insensitive manner (which is different from how most old-style wrappers work).

By default, the OSH tag for an argument is formed by prepending a hyphen to the argument's name; the oshName option can be used to override this rule.

Each argument description has a required description string. This is used when generating a usage string, and can presumably be used in the GUI.

An argument description may have some number of value descriptions, detailed below. An optional sub-argument list description may be provided, allowing additional flags associated with an argument to be accommodated.

By default, a raw argument vector may have any number of items matching a given argument description. The minOccurrences and maxOccurrences parameters may be used to restrict the number of times an argument may occur; the optional parameter is sugar for minOccurrences=0 and maxOccurrences=1.

An argument may additionally have one or more OSH alias names, allowing abbreviations, variant spellings, etc. A silent alias is not listed in a generated usage string.

The default flag indicates that this argument represents a default that is in force if this argument (and typically other related

arguments in an exclusion set with this argument) is not present. This information is put into the generated usage string, and has no other effect.

The hidden flag can be used to describe arguments that are not normally exposed to the user. Similarly, the deprecated flag can be used to indicate that an argument description exists only for back-compatibility. Hidden and deprecated argument descriptions are omitted by default from usage strings.

A deprecated flag can have an optional identifier value, which is the name of another argument that should be used instead of the deprecated argument; if the deprecated argument is used, a warning is generated indicating the preferred argument to use.

In addition to the list of argument item descriptions, an argument list desciption may have other information; this other information is tucked as sub-properties of a property called otherInfo, to minimize name collisions with argument names.

This other information includes constraints. An exclusive constraint names a set of arguments which are mutually exclusive; multiple exlusive sets may be defined. An exclusive_required constraint is similar, except one of its listed arguments must be present.

An implies constraint says that one given argument occurs, then another given argument must also be supplied.

Constraints more sophisticated than these simple ones must be implemented in the operator's C++ code that processes the resulting argItem list.

The optional description string is added to the generated usage string.

Argument Values

Each argument type, as described in an argument description, may optionally have an expected value (or multiple expected values). For example, tsort's -key argument has one required value (the field name), which can be followed by one optional value (the field type).

Each argDesc entry may have any number of

value={<value description>}

entries, to describe the value(s) that are expected to follow a particular argument's name on the command line. The order in which the value entries are listed is important, determining the order in which the values must be presented on the command line after the argument name.

```
A value description property list consists of the following
properties:

  value={type={<type>, other props}, // required
         usageName=STRING,           // required
         optional,                   // optional
         default=TL,                 // optional
         deprecated                  // optional
        }

A type must be provided; a future enhancement might be to allow
more than one type for a single value.  The available type
descriptions are described below.

The optional flag indicates that the value need not be supplied.
For a given argDesc, only its final value may be optional.

The default property can be used to specify the default value used
if this argument is not supplied; this only affects the generated
usage string.  The type's print/scan generic function is used to
read the type's literal value.

If the deprecated flag is supplied, then the value being described
will be omitted by default from generated usage strings.

The usageName string is used in generated usage strings.

The following types are available:

  type={string,
        list={STRING, STRING, ...}, // optional; list of legal values
        regexp=S,                   // optional; regexp for legal values
        case=sensitive|insensitive  // optional; default is case-insensitive
       }

  If no list or regexp are provided, than any string value is
  accepted.  If case=insensitive (default), then list matching is
  performed in a case-insensitive manner, and the regexp is
  evaluated on a copy of the string value that has been folded to
  lower case.

  type={int,                  // must be first property
        min=INT,              // optional; default is no lower limit
        max=INT,              // optional; default is no upper limit
        list={INT, INT, ...}, // optional; list of legal values; exclusive
                              //           with min/max
       }

  Integer values are 32 bits, signed.  The field value is encoded
  as a dfloat in the argument item's value=V property.

  type={float,                // must be first property
        min=FLT,              // optional; default is no lower limit
        max=FLT,              // optional; default is no upper limit
        list={FLT, FLT, ...}, // optional; list of legal values; exclusive
```

```
                              //           with min/max
        }

   Floating point values are double precision.  The field value is
   encoded as a dfloat in the argument item's value=V property.

   type={FIELDTYPE,           // must be first property
         min=TL,              // optional; default is no lower limit
         max=TL,              // optional; default is no upper limit
         list={TL, TL, ...},  // optional; list of legal values; exclusive
                              //           with min/max
         compareOptions={...} // optional; adjusts how comparisons are
                              //           done with min, max, or list values
        }

   All Orchestrate field types other than string, int, and float are
   processed in the above manner; min/max values may be specified if
   the field type supports ordered comparison; a list may be
   provided if the field type supports equality comparison.  The
   print/scan generic function is used to parse the type literal
   values TL.  The field value is encoded as a string in the
   argument item's value=V property.

   type={propList,
         elideTopBraces,      // optional
         requiredProperties={IDENT, IDENT, ...} // optional
        }

   For propList, the field value is encoded as a property list in
   the argument item's value=V property.

   type={schema,
         acceptSubField,      // optional; default is top-level fields only
         acceptVector,        // optional; default is no vectors
         acceptSchemaVar      // optional; default is no schema vars
        }

   For schema and all other value types below, the field value is
   encoded as a string in the argument item's value=V property.

   type={fieldName,
         input|output,        // required
         acceptSubField       // optional; default is top-level field only
        }

   // TBD: add a type to represent field lists (see
   // apt_framework/utils/fieldlist.h)

   type={fieldTypeName,
         list={IDENT, IDENT, ...}, // optional; lists accepted type names
         noParams             // optional; default is to accept type params
        }

  The fieldName and fieldType value types are provided so that the
  GUI can put up appropriate interfaces.  For example, for a
```

{fieldName,input} type, the GUI could put up a pick-list based on the view-adapted schema at the operator's input [this breaks down for multi-input operators-- which data set's view adapted schema does the GUI use?].

```
  type={pathName,
        canHaveHost,         // optional
        defaultExtension=STRING // optional
       }
```

Tsort Example

The following is an example of how tsort's command line usage could be described to the argument list processor:

```
{ key={value={type={fieldName, input},
              usageName="name"
             },
       value={type={fieldTypeName},
              usageName="type",
              optional,
              deprecated
             },
       subArgDesc={ ci={optional,
                        description="case-insensitive comparison"
                       },
                    cs={optional,
                        default,
                        description="case-sensitive comparison"
                       },
                    ebcdic={optional,
                            description="use EBCDIC collating sequence"
                           },
                    hash={optional,
                          description="hash partition using this key"
                         },
                    asc={oshAlias="-ascending",
                         optional,
                         default,
                         description="ascending sort order"
                        },
                    desc={oshAlias="-descending",
                          silentOshAlias="-des",
                          optional,
                          description="descending sort order"
                         },
                    sorted={optional,
                            description="records are already sorted by this key"
                           },
                    clustered={optional,
                               description="records are grouped by this key"
                              },
                    param={value={type={propList, elideTopBraces},
                                  usageName="params"
```

```
                                    },
                                optional,
                                description="extra parameters for sort key"
                               },
                          otherInfo={exclusive={ci, cs},
                                     exclusive={asc, desc},
                                     exclusive={sorted, clustered},
                                     description="Sub-options for sort key:"
                                    },
                     },
          description="specifies a sort key"
         },
     memory={value={type={int32, min=4},
                    usageName="mbytes",
                    default=20
                   },
             optional,
             description="size of memory allocation"
            },
     flagCluster={optional,
                  description="generate flag field identifying start of same-key runs
in output"
                 },
     stable={optional,
             default,
             description="use stable sort algorithm"
            },
     nonStable={silentOshAlias="-unstable",
                optional,
                description="use non-stable sort algorithm (can reorder same-key
records)"
               },
     stats={oshAlias="-statistics",
            optional,
            description="print execution statistics"
           },
     unique={oshAlias="-distinct",
             optional,
             description="keep only first record of same-key runs in output"
            },
     keys={value={type={schema},
                  usageName="keyschema"
                 },
           deprecated=key,
           maxOccurrences=1,
           description="schema specifying sort key(s)"
          },
     seq={silentOshAlias="-sequential",
          deprecated,
          optional,
          description="select sequential execution mode"
         },
     otherInfo={exclusive={stable, nonStable},
                exclusive_required={key, keys},
                description="Torrent sort operator:"
```

```
            }
   }

  Line counts: The above description is 90 lines; the old tsort.op
  wrapper that this description replaces is 410 lines.  The new tsort
  initializeFromArgs() logic is 130 lines; the original tsort logic is
  300 lines.

  The generated usage string for the above description is given next.
  The example assumes that we requested deprecated stuff as well as
  normal stuff (normally the deprecated items would be suppressed from
  the usage string):

     Torrent sort operator:

     -key             -- specifies a sort key; 1 or more
        name          -- input field name
        type          -- field type; optional; DEPRECATED

        Sub-options for sort key:

        -ci           -- case-insensitive comparison; optional
        -cs           -- case-sensitive comparison; optional; default
        -ebcdic       -- use EBCDIC collating sequence; optional
        -hash         -- hash partition using this key; optional
        -asc or -ascending
                      -- ascending sort order; optional; default
        -desc or -descending
                      -- descending sort order; optional
        -sorted       -- records are already sorted by this key; optional
        -clustered    -- records are grouped by this key; optional
        -param        -- extra parameters for sort key; optional
           params     -- property=value pair(s), without curly braces

        (mutually exclusive: -ci, -cs)
        (mutually exclusive: -asc, -desc)
        (mutually exclusive: -sorted, -clustered)

     -memory          -- size of memory allocation; optional
        mbytes        -- int32; 4 or larger; default=20

     -flagCluster     -- generate flag field identifying start of same-key runs in
output; optional

     -stable          -- use stable sort algorithm; optional; default

     -nonStable       -- use non-stable sort algorithm (can reorder same-key
records); optional

     -stats or -statistics
                      -- print execution statistics; optional

     -unique or -distinct
                      -- keep only first record of same-key runs in output; optional
```

```
    -keys            -- schema specifying sort key(s); optional; DEPRECATED: use
-key instead
       keyschema     -- string

    -seq             -- select sequential execution mode; optional; DEPRECATED

    (mutually exclusive: -stable, -nonStable)
    (mutually exclusive: -key, -keys; one of these must be provided)
```

Of course, the GUI will want to present this information in its own appropriate fashion.

Error Processing

When an argument description property list is used to initialize an argument processor object, any errors found in the description are placed onto a supplied error log object.

When a raw argument vector is being processed, any command line tokens that do not correspond to an argument description are taken to be error tokens. An error message is generated for each error token, with the arg token number and text being included in the error message. The error token is discarded and the next argument token is attempted to be processed according to the argument description.

In a sub-argument list, an unrecognized argument token terminates the sub-argument list (at which point any constraints are checked), and pops processing back up to the enclosing argument list.

When an argument token is seen that matches one of the argument descriptions, subsequent argument tokens are processed according to that argument description. Any local discrepancies (bad value, bad sub-argument list, etc.) are reported along with the argument token(s) making up the argument being processed, along with usage-style information relating to that argument.

When an argument list is terminated, the constraints are checked (occurrences restrictions, exclusivity and implies relationships) and any violations reported with appropriate contextual information (the argument instance(s) involved, appropriate usage-style information describing the violated constraint(s)).

When one or more errors are reported, only argument tokens and usage information specific to the errors at hand are included in the error message(s). Whether or not the entire usage string is generated and issued in a summary message is up to the client, as is the responsibility of identifying the operator (or other context) in which the argument processing error occurred.

GUI Discussion

Although this argument processing facility is designed for use with

OSH, it can also be used by the GUI to help determine the content of the control panels used to configure each kind of operator.

However, the GUI will require much more information than is present in the above specifications. For example, how does the GUI know what options to tie together in a "radio button" group? Knowing that a set of options is mutually exclusive is not quite enough, because the fact that two options are mutually exclusive does not necessarily mean that they should be presented in a radio button group.

Another example is that the description above takes the form of a 1-dimensional list of definitions, but a GUI presents information using at least 2 dimensions (one can think of a tabbed dialog as representing a 3-dimensional grouping of controls). Clearly the information above is necessary, but not sufficient, to drive the GUI's operator control panels.

At some point, it will become necessary to augment the argument descriptions with GUI-specific information, mostly for layout but perhaps also for certain semantic indications. This GUI-specific information might be authored separately from the operator-based argument list description, with the argument names being used to "join" the GUI-specific information to the operator-based argument description.

Or, the GUI-specific information might simply be added to the operator-based agument list description via new properties. To keep this option open, all of the property lists presented above should consider the property name "gui" to be reserved for future use.

Geek Speak

This facility can be seen as a specialized parser generator: the argument description is a grammar that describes a recursive-descent parser that processes a raw argument vector and produces a parse tree result.
*/

```
#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

class APT_PropertyList;
class APT_String;
class APT_ErrorLog;

class APT_ArgvProcessorRep;

class APT_ArgvProcessor
{
public:
  APT_ArgvProcessor();
  ~APT_ArgvProcessor();
```

```
APT_ArgvProcessor(const APT_PropertyList& description, APT_ErrorLog&);

void setDescription(const APT_PropertyList& description, APT_ErrorLog&);
APT_PropertyList description() const;

APT_PropertyList processArglist(int argc, const char* const* argv,
                                APT_ErrorLog&) const;
APT_PropertyList processArglist(int argc, const APT_String* argv,
                                APT_ErrorLog&) const;
/*
   effect    Processes the raw argument vector according to the
             description().
             If successful, a property list will be returned
             describing the various argument items encountered, and
             no errors will be written into the error log (but
             info/warning messages can be generated, such as to flag
             deprecated argument usages).
             If unsuccessful, one or more errors will be written
             into the error log, and a property list containing only
             the successfully processed argument items will be
             returned.  Note that argument items involved in a
             'exclusive' or 'implies' constraint violation are
             retained in the returned list, but argument items that
             are invalid for a value constraint violation (such as
             min, max, etc.) are not retained.
   note      Only argv[1]..argv[argc-1] are examined by this
             routine.  argv[0] is ignored.
             The argv should be allocated to argc elements; there is
             no need for an extra null argv element.
   requires  argc >= 1
             argv non-null
             argv elements non-null, null terminated strings.
             This APT_ArgvProcessor must have been successfully
             initialized with a description.
*/

APT_PropertyList processArglist(const APT_PropertyList& argv,
                                APT_ErrorLog&) const;
/*
  effect     See processArglist above. The only difference is that
             instead of getting the arguments from an argc/argv
             vector, they are taken from a property list containing
             a sequence of "arg" items, like this:

             {arg="someOp", arg="-doSomethingUseful", arg="no"}

             (instead of an arg vector pointing to the three
             strings "someOp", "-doSomethingUseful" and "no").
             As above, the first item in the property list
             is ignored.

             This is intended as a temporary migration aid
             until argument processing is built in to the framework.
             Expect this to disappear in a future release.
```

```
  */


  APT_String oshUsage(bool showHidden=false, bool showDeprecated=false,
                      int indent=0, int remarkCol=24) const;
  /*
     effect    Returns an OSH usage string describing the various
               arguments expected by this argument processor.
     note      The usage() is synthesized from the description() and
               may be somewhat verbose.
     requires  This APT_ArgvProcessor must have been successfully
               initialized with a description.
  */

private:
  APT_ArgvProcessor(const APT_ArgvProcessor&);
  APT_ArgvProcessor& operator= (const APT_ArgvProcessor&);

  APT_ArgvProcessorRep* rep_;
};

#endif // APT_ARGVCHECK_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ASSERT_H
#define APT_ASSERT_H

#ifndef APT_ERRIND_H
#include <apt_util/errind.h>
#endif

class APT_StackTrace;

#include <iostream.h>           // I hate to include these, but
                                // APT_REQUIRE2() makes them
#include <strstream.h>          // necessary. Better to include them here than
                                // to nag everyone to explicitly include them.

// New fatal error message output class.  Needs to deal with the macros:
//      APT_USER_REQUIRE        Includes module id & error index.
//                              Prefix with "Requirements failure: "
//      APT_USER_REQUIRE_LONG
//      APT_ASSERT              Out module id & index; input file & line.
//                              Output: "Assertion failure: !expr: File %f Line %l"
//                              Implement with APT_MSG_ASSERT
//      APT_MSG_ASSERT          Add message after "!expr: "
//      APT_DETAIL_FATAL        Includes module id, error index, & message.
//                              Just output the message.
//      APT_DETAIL_FATAL_LONG

class APT_FatalPath
{
  public:
    static int pureAssertion(const char *expr, const char *file, int line);
    static void msgAssertion(const char *expr, ostrstream &msg,
                             const char *file, int line);
    static void requireFail(ostrstream &msg, APT_Error::SourceModule &mod,
                            APT_Error::ErrorIndex index);
    static void fatalFail(ostrstream &msg, APT_Error::SourceModule &mod,
                          APT_Error::ErrorIndex index);
    // effect   Reports errors of the various types.  If they are given
    //          a stream argument, they stick an ends on it, grab the
    //          string, and free the storage.  This pretty much means that
    //          they are only good for supporting the below macros.
    //          If the environment variable
    //          APT_ASSERTION_FAILURE_DOES_NOT_ABORT is set, the assertion
    //          failures return (the require and fatal fails still exit).

    typedef void (*hook)();

    static void addMessageHook(hook);
    /*
       effect   Adds a callback to be called before the standard
                failure message and stack trace is generated.  The
```

```
                message callbacks are executed in the order they were
                added.
        note    This routine is usable at STI time.
    */

    static hook setFatalPathHook(hook fatalPathHook);
    // effect   Sets a function pointer to be called whenever control
    //          goes through this routine.  Defaults to null; if hook
    //          is null, no call is done.  All message output is done
    //          before this hook is called.  Returns old value of hook.
    // notes    May be used to arrange for an exception to be thrown
    //          or for the process to die on a signal rather than the
    //          default behavior (which is to call _exit(1)).
    //          If the fatalPathHook function returns control, the
    //          default _exit(1) behavior occurs.
    //          This routine is usable at STI time.

    static void setStackTrace (APT_StackTrace* stackTrace);
    // effect  Set a stack trace object that can be used to create
    //         stack trace upon fatal error.  Defaults to null; if
    //         null, no stack trace will be created.
    // note    This routine is usable at STI time.

  private:
    static void messageHooks();
    // effect   Called on all fatal path error messages.  Calls the
    //          hook(s) set by addMessageHook()

    static void fatalPathHook();
    // effect   Called on all fatal path error messages.
    //          Calls the hook set by setFatalPathHook (if any); else
    //          a nop.  May also be used for debugging by setting a
    //          breakpoint here.

    static hook *messageHooks_;
    static int numMessageHooks_;

    static hook fatalPathHook_;

    static APT_StackTrace* stackTrace_;
};

extern void
APT_cleanup_exit(int status);

#define APT_USER_REQUIRE_LONG(predicate_, module_, errorIndex_, message_) \
  do {                                                                    \
      if (! (predicate_)) {                                               \
          ostrstream _stream_;                                            \
          _stream_ << message_;                                           \
          APT_FatalPath::requireFail(_stream_, module_, errorIndex_);     \
      }                                                                   \
  } while (false)
```

```
#define APT_DETAIL_FATAL_LONG(module_, errorIndex_, message_)   \
  do {                                                          \
      ostrstream _stream_;                                      \
      _stream_ << message_;                                     \
      APT_FatalPath::fatalFail(_stream_, module_, errorIndex_); \
  } while (false)


#define APT_MSG_ASSERT(e_, msg_)                                        \
  do {                                                                  \
      if (!(e_)) {                                                      \
          ostrstream _stream_;                                          \
          _stream_ << msg_;                                             \
          APT_FatalPath::msgAssertion(#e_, _stream_, __FILE__, __LINE__); \
      }                                                                 \
  } while (false)

#define APT_ASSERT(e_)  \
    ((e_) ? 1 : APT_FatalPath::pureAssertion(#e_, __FILE__, __LINE__))

#define APT_USER_REQUIRE(predicate_, errorIndex_, message_)             \
    APT_USER_REQUIRE_LONG(predicate_, APT_localErrorSourceModule,       \
                          errorIndex_, message_)

#define APT_DETAIL_FATAL(errorIndex_, message_) \
    APT_DETAIL_FATAL_LONG(APT_localErrorSourceModule, errorIndex_, message_)

/* by ending include guard here, subsequent stuff can be turned on and
   off within a compilation unit by changing APT_NODEBUG and re-including
   this header file. */

#endif // APT_ASSERT_H

#undef APT_ASSERT_DEBUGONLY
#undef APT_MSG_ASSERT_DEBUGONLY

#ifdef APT_NODEBUG
#  define APT_ASSERT_DEBUGONLY(e) do {} while (0)
#  define APT_MSG_ASSERT_DEBUGONLY(e, msg) do {} while (0)
#else
#  define APT_ASSERT_DEBUGONLY(e) APT_ASSERT(e)
#  define APT_MSG_ASSERT_DEBUGONLY(e, msg) APT_MSG_ASSERT(e, msg)
#endif
```

```
/* -*-Mode: C-*- */
/* Copyright (c) 1995-1997 Torrent Systems, Inc. All rights reserved.  */

#ifndef APT_CONDITION_H
#define APT_CONDITION_H

/*
  Code to support conditional compilation
  This file may be #include'd by both C and C++ code.  In particular, this
  means that all comments must be C-style rather than C++-style.
  Furthermore, any pointers to structs in function prototypes must
  have previously #include'd the necessary header file.
*/

/* Use these flags to conditionalize source code.
   eventually we'll add more precise conditionalization flags
   but you should use the most generic flag, e.g. don't use __GPLUSPLUS__
   to discern whether it's SUN or not.

   With respect to version flags. Use flags that you know work. Don't
   assume that it works on all __AIX_3__ if it works on __AIX_3_2__

   Compilers: __GPLUSPLUS__, __XLC__, __SUNPRO__
   Compiler versions: ... someday when needed....
   Operating Systems: generic __SUN__ (meaning solaris), __AIX__
   OS Major Versions: __AIX_3__, __AIX_4__
   OS Detailed Versions: __AIX_3_2__, __AIX_4_1__


   The above flags are developed using flags pre-defined by the compiler
   on various platforms:

   _AIX: defined by xlC on all (?) aix versions
   _AIX32: defined by xlC on aix 3.2 (and higher?)
   _AIX41: defined by xlC on aix 4.1 (and higher?)
   __xlC__: defined by xlC with a value encoding the compiler version
   __SUNPRO_CC: defined by SunPro C++ with a value encoding the compiler
                version
   __GNUC__: defined by gcc and g++
   __HP_aCC: defined by HPUX aCC releases A.01.15 and later
   __alpha: defined by DEC cxx
   __hpux: defined by the aCC and cc compilers on HPUX
   __cplusplus: defined by all (?) C++ compilers

   The compiler does not appear to define anything for AIX 4.2???
   The build system defines _AIX42 on AIX 4.2 systems
   The build system defines _AIX43 on AIX 4.3 systems
*/

/* currently six platforms: sun, aix, linux, ncr, hpux and OSF1. */
```

```
#ifdef linux
#  undef __LINUX__
#  define __LINUX__
#elif defined(_ATT4) && defined(_nst) && defined(__HIGHC__)
#  undef __NCR
#  define __NCR__
#elif defined(_AIX)
#  undef __AIX__
#  define __AIX__
#elif defined(__HP_aCC) || defined(__hpux)
#  undef __HPUX__
#  define __HPUX__
#elif defined(__alpha)
#  undef __OSF1__
#  define __OSF1__
#elif defined(__sun)
#  undef __SUN__
#  define __SUN__
#else
#error "Platform Not Recognized"
#endif

/* We want __AIX__ as our flag */
#ifdef __AIX__
   /* This compound predicate required because on AIX 4.1's xlC compiler */
   /* _AIX32 is STILL DEFINED! (ugh) */
#  if defined(_AIX43)
#    undef __AIX_4__
#    define __AIX_4__
#    undef __AIX_4_3__
#    define __AIX_4_3__
#    undef __AIX_4_2__
#    define __AIX_4_2__
#  elif defined(_AIX42)
#    undef __AIX_4__
#    define __AIX_4__
#    undef __AIX_4_2__
#    define __AIX_4_2__
#  elif defined(_AIX41)
#    undef __AIX_4__
#    define __AIX_4__
#    undef __AIX_4_1__
#    define __AIX_4_1__
#  elif defined(_AIX32) && !defined(_AIX41)
#    undef __AIX_3__
#    define __AIX_3__
#    undef __AIX_3_2__
#    define __AIX_3_2__
#  else
#  error Unknown version of AIX
#  endif /* defined(_AIX32) && !defined(_AIX41) */
```

```
#  ifdef __xlC__
#    undef __XLC__
#    define __XLC__
#  endif
#endif /* _AIX */

#ifdef __SUN__
#  if defined(__SunOS_5_5) || defined(__SunOS_5_5_1)
#    undef __SOLARIS_2__
#    define __SOLARIS_2__
#    undef __SOLARIS_2_5__
#    define __SOLARIS_2_5__
#  elif defined(__SunOS_5_6) || defined(__SunOS_5_6_1)
#    undef __SOLARIS_2__
#    define __SOLARIS_2__
#    undef __SOLARIS_2_5__
#    define __SOLARIS_2_5__
#    undef __SOLARIS_2_6__
#    define __SOLARIS_2_6__
#  elif defined(__SunOS_5_7) || defined(__SunOS_5_7_1)
#    undef __SOLARIS_2__
#    define __SOLARIS_2__
#    undef __SOLARIS_2_5__
#    define __SOLARIS_2_5__
#    undef __SOLARIS_2_6__
#    define __SOLARIS_2_6__
#    undef __SOLARIS_2_7__
#    define __SOLARIS_2_7__
#  elif defined(__SunOS_5_8) || defined(__SunOS_5_8_1)
#    undef __SOLARIS_2__
#    define __SOLARIS_2__
#    undef __SOLARIS_2_5__
#    define __SOLARIS_2_5__
#    undef __SOLARIS_2_6__
#    define __SOLARIS_2_6__
#    undef __SOLARIS_2_7__
#    define __SOLARIS_2_7__
#    undef __SOLARIS_2_8__
#    define __SOLARIS_2_8__
#  else
#    error Unknown version of Solaris
#  endif
#endif /* __SUN__ */

#ifdef __NCR__
#define HAVE_RWMATH
/* TBD: grovel NCR system versions */
#endif /* __NCR__ */
```

```
#if defined(__GNUC__) && defined(__cplusplus)
// #define HAVE_RWMATH
#  define __GPLUSPLUS__
#endif

#if defined(__SUN__) && defined(__SUNPRO_CC)
#  define __SUNPRO__
#endif


/* exceptions? */

#if defined(__XLC__)
#  define __EXCEPTIONS__
#endif

#if defined(__NCR__)
#  define __EXCEPTIONS__ 1
#endif

#if defined(__SUNPRO__)
#  define __EXCEPTIONS__
#  define __NEW_FAILURE_EXCEPTION__
#endif

#if defined(__HPUX__)
#  define __EXCEPTIONS__
#endif

#if defined(__OSF1__)
#  define __EXCEPTIONS__
#if defined(__cplusplus) && !defined(__EXCEPTIONS)
// Did you disable exceptions?
#error __EXCEPTIONS not provided by compiler
#endif
#endif


/* bool? */

#if defined(__GPLUSPLUS__) || \
    defined(__HPUX__)   || \
    defined(__OSF1__)
#define __BOOL__
#endif


/* SunPRO 5 has bool; 4.2 apparently doesn't */
#if defined(__SUNPRO_CC)
#if __SUNPRO_CC >= 0x500
#define __BOOL__
```

```
#endif
#endif


/* malloc() pointer alignment */

#if defined(__NCR__)
#define __MALLOC_ALIGN__ 4
#else
#define __MALLOC_ALIGN__ 8      /* most platforms do this */
#endif


/* standard explicit template instantiation syntax? */

#if defined(__NCR__)
/* no explicit instantiation syntax needed */
#  define __AUTOMATIC_TEMPLATE_INSTANTIATION__ 1
#endif

#if defined(__GPLUSPLUS__)
#  define __SHOULD_INSTANTIATE_TEMPLATES__ 1
#  define SHOULD_INSTANTIATE_TEMPLATES 1
#  define __STANDARD_TEMPLATE_INSTANTIATION__ 1
#endif

#if defined(__SUNPRO__)
#  define __SHOULD_INSTANTIATE_TEMPLATES__ 1
#  define SHOULD_INSTANTIATE_TEMPLATES 1
#  define __STANDARD_TEMPLATE_INSTANTIATION__ 1
#endif

#if defined(__HPUX__)
#  define __SHOULD_INSTANTIATE_TEMPLATES__ 1
#  define SHOULD_INSTANTIATE_TEMPLATES 1
#  define __STANDARD_TEMPLATE_INSTANTIATION__ 1
#endif

#if defined(__OSF1__)
#  define __SHOULD_INSTANTIATE_TEMPLATES__ 1
#  define SHOULD_INSTANTIATE_TEMPLATES 1
#endif

/* conditional compilation feature flags */

/*
   If APT_INT32_IS_NOT_INT is defined, overloads of "int" and
   "APT_Int32" are distinguished, and both need to be provided.  I
   don't think any platforms fall into that category.

   #define APT_INT32_IS_NOT_INT
```

```
*/

#if defined (__LINUX__)
#  define HAVE_LONG_LONG
#  define UNALIGNEDACCESS 1
#  define LITTLEENDIAN 1
#  define HAVE_SIGACTION 1
#  define HAVE_SA_HANDLER 1
#  define HAVE_SIGTRAP 1
#  define HAVE_GETRUSAGE 1
#  define HAVE_TIMES 1
#  define HAVE_STATFS 1
#  define HAVE_SYS_VFS_H 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
#  define APT_SERIALIZE_TIME_T
#include <strings.h>

#elif defined (__SUN__)
#  define HAVE_LONG_LONG
#  define BIGENDIAN 1
#  define HAVE_SIGACTION 1
#  define HAVE_SA_SIGACTION 1
#  define HAVE_SYS_SELECT_H 1
#  define HAVE_STROPTS_H 1
#  define HAVE_SIGSYS 1
#  define HAVE_PIOCUSAGE 1
#  define HAVE_TIMES 1
#  define HAVE_STATVFS 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
#  define APT_SERIALIZE_TIME_T

#include <strings.h>

#ifdef __cplusplus
extern "C" int gettimeofday (struct timeval *, void *);
#else
extern int gettimeofday (struct timeval *, void *);
#endif

#if (_POSIX_C_SOURCE - 0 >= 199506L) || defined(_POSIX_PTHREAD_SEMANTICS)
#define POSIX_1003_1C
#endif

#elif defined (__HPUX__) /* Need to check this stuff out... */
#  define HAVE_LONG_LONG
#  define BIGENDIAN 1
#  define HAVE_SIGACTION 1
#  define HAVE_SA_SIGACTION 1
/* #  define HAVE_SYS_SELECT_H 1 */
```

```
#  define HAVE_STROPTS_H 1
#  define HAVE_SIGSYS 1
/* #  define HAVE_PIOCUSAGE 1 */
#  define HAVE_TIMES 1
#  define HAVE_STATVFS 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
#  define HAVE_ANSI_BAD_ALLOC_NEW 1
#  define APT_SERIALIZE_TIME_T
#include <strings.h>

#if defined(_REENTRANT) && !defined(_PTHREADS_DRAFT4)
#define POSIX_1003_1C
#endif

#elif defined (__OSF1__)
#  define LITTLEENDIAN 1
#  define HAVE_SIGACTION 1
#  define HAVE_SA_SIGACTION 1
#  define HAVE_SYS_SELECT_H 1
#  define HAVE_STROPTS_H 1
#  define HAVE_SIGSYS 1
/* #  define HAVE_PIOCUSAGE 1 */
#define HAVE_GETRUSAGE
#  define HAVE_TIMES 1
#  define HAVE_STATVFS 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
#  define HAVE_ANSI_BAD_ALLOC_NEW 1
#  define EXPLICIT_COPY_FROM_SCOPE 1
#  define LONG_IS_64_BITS 1

#include <strings.h>

#elif defined (__NCR__)
#  define HAVE_LONG_LONG
#  define UNALIGNEDACCESS 1
#  define LITTLEENDIAN 1

/* I don't know these... smr */
#  define HAVE_SIGACTION 1
#  define HAVE_SA_HANDLER 1
/* #  define HAVE_SA_SIGACTION 1        // seems we don't have this */
#  define HAVE_SYS_SELECT_H 1
#  define HAVE_STROPTS_H 1
#  define HAVE_SIGSYS 1
/* #  define HAVE_PIOCUSAGE 1   // seems we don't have this */
#  define HAVE_TIMES 1
#  define HAVE_STATVFS 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
```

```
#  define APT_SERIALIZE_TIME_T

/* ncr's strings.h is evil: */
#include <strings.h>
#undef bcmp
#undef bcopy
#undef bzero
#undef index
#undef rindex

/* ncr's resource.h is equally bogus: */
#include <sys/time.h>

#ifdef __cplusplus
extern "C" {
#endif

struct rusage {
        struct timeval ru_utime;        /* user time used */
        struct timeval ru_stime;        /* system time used */
        long    ru_maxrss;
#define ru_first        ru_ixrss
        long    ru_ixrss;               /* XXX: 0 */
        long    ru_idrss;               /* XXX: sum of rm_asrss */
        long    ru_isrss;               /* XXX: 0 */
        long    ru_minflt;              /* any page faults not requiring I/O */
        long    ru_majflt;              /* any page faults requiring I/O */
        long    ru_nswap;               /* swaps */
        long    ru_inblock;             /* block input operations */
        long    ru_oublock;             /* block output operations */
        long    ru_msgsnd;              /* messages sent */
        long    ru_msgrcv;              /* messages received */
        long    ru_nsignals;            /* signals received */
        long    ru_nvcsw;               /* voluntary context switches */
        long    ru_nivcsw;              /* involuntary " */
#define ru_last         ru_nivcsw

};

extern int getrusage(int, struct rusage *);

#ifdef __cplusplus
}       /* extern "C" */
#endif

#elif defined (__AIX__)
#  define HAVE_LONG_LONG
#  define UNALIGNEDACCESS 1
#  define BIGENDIAN 1
#  define HAVE_SIGACTION 1
#  define HAVE_SA_HANDLER 1
```

```
#  define HAVE_SYS_SELECT_H 1
#  define HAVE_STROPTS_H 1
#  define HAVE_SIGSYS 1
#  define HAVE_SIGTRAP 1
#  define HAVE_GETRUSAGE 1
#  define HAVE_TIMES 1
#  define HAVE_STATFS 1
#  define HAVE_SYS_STATFS_H 1
#  define HAVE_GETTIMEOFDAY2 1
#  define HAVE_RLIMIT 1
#  if defined(_AIX42)
#    define APT_SERIALIZE_TIME_T
#  endif
#include <strings.h>

#if ((_XOPEN_SOURCE + 0)==500) && !defined(_UNIX95)
#  define POSIX_1003_1C
#endif

#include <sys/statfs.h>
/*
 Unfortunately the header for this function changed between 4.3 and 4.1 & 4.2
 It doesn't have a prototype in 4.1 and 4.2, so the first argument can be
 a const (as it should be.)  4.3 has a prototype, but it isn't const.  This
 causes an error because it sees two definitions of statfs (the OS's non-const
 and the orchestrate's const version.
*/

#ifdef __cplusplus
extern "C" {
#endif

#ifdef __AIX41__
extern int readv (int, struct iovec *iov, int);
#endif

#  if defined(_AIX43)
extern int statfs (char *, struct statfs *);
#  elif defined(_AIX42)
extern int statfs (const char *, struct statfs *);
#  elif defined(_AIX41)
extern int statfs (const char *, struct statfs *);
#  else
#    error "This version of AIX handles statfs in an unknown way."
#  endif


#ifdef __cplusplus
}       /* extern "C" */
#endif
```

```
#endif /* Platform-specific flaggage */

/*
  The slippery slope towards ANSI C++.
*/

#if defined(__USE_REINTERPRET_CAST__)
#define REINTERPRET_CAST(type_, expr_) \
reinterpret_cast<type_>(expr_)
#else
#define REINTERPRET_CAST(type_, expr_) \
(type_)(expr_)
#endif

#if defined(__USE_CONST_CAST__)
#define CONST_CAST(type_, expr_) \
const_cast<type_>(expr_)
#else
#define CONST_CAST(type_, expr_) \
(type_)(expr_)
#endif

/*
   pointer to int32 conversion

   There are places (like hash functions) where it's reasonable
   to convert a pointer to an 32-bit integer. Provide macros to
   do this in a controlled fashion.

   ??? hash the value to enable spotting problems on 32-bit machines?
*/

#if defined(__LINUX__)  || \
    defined(__NCR__)    || \
    defined(__AIX__)    || \
    defined(__HPUX__)   || \
    defined(__SUN__)    || \
    defined(__OSF1__)
#define APT_POINTER_TO_INT32(value) ((int) ((long) value))
#define APT_POINTER_TO_UINT32(value) ((unsigned) ((unsigned long) value))
#else
#error "Macros to convert pointers to 32-bit integers not defined"
#endif

#endif /* APT_CONDITION_H */
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_DATE_H
#define APT_DATE_H

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

// This is the header file for the APT_Date class.  This class represents
// a date in a general fashion.  It provides functions for initializing
// objects from several printed date forms, doing arithmetic on dates
// (including comparison) and printing them out.

// It is loosely based on a date class developed by M. A. Sridhar,
// which is based on an algorithm published in CACM, Aug 1963
// (Algorithm 199).  His copyright notice follows:

//     Copyright (C) 1995, M. A. Sridhar
//
//
// This software is Copyright M. A. Sridhar, 1995. You are free
// to copy, modify or distribute this software  as you see fit,
// and to use  it  for  any  purpose, provided   this copyright
// notice and the following   disclaimer are included  with all
// copies.
//
//                    DISCLAIMER
//
// The author makes no warranties, either expressed or implied,
// with respect  to  this  software, its  quality, performance,
// merchantability, or fitness for any particular purpose. This
// software is distributed  AS IS.  The  user of this  software
// assumes all risks  as to its quality  and performance. In no
// event shall the author be liable for any direct, indirect or
// consequential damages, even if the  author has been  advised
// as to the possibility of such damages.

// This class does not perform correct year/month/day calculations on
// dates before September 14th, 1752, although dates back to about
// year 1 can be represented by this class.

// A note on terminology; we distinguish a Julian Date (which is a
// specified by giving year and day-of-year) from a julian day count
// (which is the number of days from 4713 BCE January 1, 12 hours GMT
// (Julian proleptic Calendar)).

// This class can parse a limited set of string formats, all representing
// the parts of their data as numeric.  Specifically, any format string
// constructed as follows:
//
// %dd                  Two digit days; must be zero padded if < 10
```

```
// %mm                  Two digit month; must be zero padded if < 10
// %mmm                 3 character month representation, for example Jan,
//                      Feb, etc.  Note that this representation is case
//                      insensitive.  When exporting data in this format,
//                      the string representation will always be lower case.
// %<full year>yy       Two digit year, parse with year cutoff == <full year>
//                      Must be zero padded if < 10
// %yy                  Two digit year; parsed with yearcutoff ==
//                      class default year cutoff.
// %yyyy                Full four digit year.
// %ddd                 Day of year in three digit form.  Must be zero
//                      padded if < 100.  On input,
//                      this is incompatible with either %dd or %mm.
//
// All other characters match anything on input, and are output as
// themselves on output.  ISO format would thus be "%yyyy-%mm-%dd".
//
// The "year cutoff" is the year that defines the beginning of the
// century in which all two-digit years will be considered to fall.
// So for instance, if the year cutoff is 1930, the two digit year
// "31" would be considered to be 1931, but the two digit year "29"
// would be considered to be 2029.
//
// A possible future expansion is to have the default year cutoff setable;
// there will be a static setYearCutoff() member function on the
// class.  There is currently a static getYearCutoff member function;
// it returns 1900.  We do not currently have the setter for this
// value since there is no simple way to make sure that that value is
// propagated to the players.
//
// Note that all of these are fixed width formats which require zero padding.
// A possible future enhancement is to allow on both input and output both
// space padded years, months, and days, and no padding at all.  These
// additions might make parsing somewhat less efficient.

// Another possible future expansion might be to have a "fast parse"
// that didn't do the error checking to make sure that it was passed a
// valid date string.


class ostream;

class APT_Date
{
  // Persistence support.

friend APT_Archive &operator ||(APT_Archive &ar, APT_Date &d);

  public:

    // Default copy constructor and assignment operator is ok;
    // representation is bitwise copyable.

    // We specify the copy constructor to attempt to workaround an xlC bug.
```

```
   APT_Date(const APT_Date &d) : days_(d.days_) {}

   // Default destructor is ok; nothing to destroy.  Note that we
   // very much don't want this object to have a vtbl, so putting a
   // virtual destructor in place isn't feasable.

   /*
    * Construction and setting.
    */

   APT_Date();
   // effect   Creates a date object for which isValid() is false.
   //          A valid date value must be assigned to this object
   //          before its value can be used.

   APT_Date(const char *dateString,
            const char *formatString = "%yyyy-%mm-%dd");
   // effect   Creates a date object from parsing the given string.
   //          If the string doesn't match the given format, an invalid
   //          date object is created.  The default format is
   //          ISO dates.
   // requires formatString must be a valid format (see above).
   // note     The string need not be null terminated; the
   //          formatString fully specifies the parsing without
   //          regard to nulls.

   APT_Date(int year, int month, int day);
   // effect   Constructs a date object with the given year, month,
   //          and day.  Note that no year cutoff is applied to the
   //          year.  If the month or day provided is invalid, or the
   //          date represented is not within our representable range,
   //          an invalid date object is produced.

   void set(const char *dateString,
            const char *formatString = "%yyyy-%mm-%dd");
   // effect   Sets this date to the date represented in dateString
   //          A parse error will result in this date being set invalid.
   // requires The format string must be valid.
   // note     The string need not be null terminated; the
   //          formatString fully specifies the parsing without
   //          regard to nulls.

   void setFromJulianDate(int year, int dayOfYear);
   // effect   Sets the date object to the date with the given year
   //          and day of year.  If the date is outside our
   //          representable range or the day provided is invalid, an
   //          invalid date is created.

   void setFromJulianDay(APT_UInt32 julianDay);
   // effect   Sets the date object to the date with the given julian
   //          day count.  If the day is outside our
   //          representable range, an inavlid date is created.

   /*
```

```
    * Accessors.
    */

   bool isValid() const;
   // effect   Returns true if the object represents a valid
   //          date; false otherwise.

   APT_String asString(const char *format = "%yyyy-%mm-%dd") const;
   // effect   Returns a string representation of the date, formatted
   //          according to the passed argument.  An
   //          invalid date will be represented by a string of '*'s

   APT_UInt32 julian() const;
   // effect   Returns the julian day count (days since 4714 BCE November
   //          24, 12 hours GMT Gregorian proleptic).
   // requires isValid() == true

   int year() const;
   int month() const;
   int day() const;
   // effect   Returns the specified portion of the date.
   // requires isValid() == true

   int dayOfYear() const;
   /*
      effect   Returns the day of the year, where Jan 1 is day 1.
      note     The returned value will be on the range 1-366,
               inclusive (Dec 31 of a leap year is day 366).
   */

   int weekOfYear() const { return (dayOfYear()-1)/7 + 1; }
   /*
      effect   Returns the week number, where days Jan 1 through Jan
               7 are considered to be week 1.
      requires isValid() == true
   */

   int weekday() const;
   // effect   Returns the day of week (Sunday == 0) for the date
   //          represented by this object.
   // requires isValid() == true

   APT_Date previousWeekday(int dayOfWeek) const;
   // effect   Returns the latest date that is the given day of week
   //          and is earlier than or the same as the object date.
   //          dayOfWeek is calibrated with Sunday == 0.
   //          If isValid() == false, the resulting date will also be
   //          invalid.
   // requires 0 <= dayOfWeek < 7

   APT_Date nextWeekday(int dayOfWeek) const;
   // effect   Returns the earliest date that is the given day of week
   //          and is later than or the same as the object date.
   //          dayOfWeek is calibrated with Sunday == 0.
```

```
    //          If isValid() == false, the resulting date will also be
    //          invalid.
    // requires 0 <= dayOfWeek < 7


    /*
     * Arithmetic member functions and operators.  The
     * other arithmetic functions are implemented at the end of this
     * file as inline functions in terms of these member functions;
     * see the list at the end of this file.
     *
     * If the result of an arithmetic function would produce a date outside
     * of the range of representable dates, the date produced is invalid.
     */

    APT_Date &operator+=(APT_Int32 days);
    // effect   Adds the specified number of days to the object.
    //          If isValid() == false, the resulting date will also be
    //          invalid.

    APT_Date &operator-=(APT_Int32 days);
    // effect   Subtracts the specified number of days to the object.
    //          If isValid() == false, the resulting date will also be
    //          invalid.

friend APT_Int32 operator -(const APT_Date &op1, const APT_Date &op2);
    // effect   Subtracts the two dates, returning the number of days
    //          between them.
    // requires Both dates must have isValid() == true

    /*
     * Note that we do not support functions that give differences
     * between dates in terms of years, months, and days.  These
     * functions may be built as needed out of daysInYear and
     * daysInMonth, or they may be added at some future point.
     */

    static int compare(const APT_Date &op1, const APT_Date &op2);
    // effect   Returns -1, 0, or +1 according to whether op1 is less
    //          than, equal to, or greater than op1.
    // requires Both dates must have isValid() == true

    /*
     * Miscellaneous stuff.
     */

    APT_UInt32 hash() const;
    // effect   Returns an Int32 that is well suited for hashing
    //          functions (i.e. unique or near unique for a
    //          specific date).
    // requires isValid() == true

    static APT_Date smallestDate();
    // effect   Returns the smallest valid date that may be
```

```
//          representable within an APT_Date object.

static APT_Date today();
// effect   Returns a date representing todays date.  Specifically,
//          this is the date at the current instance within the local
//          timezone.

static int yearCutoff();
// effect   Returns the year cutoff in effect for this class.
//          The default is 1900.

// static void      setYearCutoff(APT_Int32 yearCutoff);
// effect   Sets the year cutoff for this class.
//          This function is currently unimplemented as there is no
//          easy way to propagate the value from conductor to players.

static int daysInMonth(int month, int year);
// effect   Returns the number of days in the given month of the
//          given year.
// requires The first day of that month must be representable in
//          our date format.

static int daysInYear(int year);
// effect   Returns the number of daysin the given year.
// requires The first day of that year must be representable in
//          our date format.

static bool isFormatValid(const char *format);
// effect   Returns true if format points to a valid format, false
//          otherwise.  The valid formats are specified above.

/*
 * Support for compiled parsing and printing.
 */
class ParseObject {
  public:
    ParseObject(const char *format);
    // effect       Creates a parse object that describes how to parse
    //              the given format in a "compiled" form; this object
    //              may be used to parse multiple instances of the
    //              given format.
    // requires     format must be a valid format string (see above).

    ParseObject();
    // effect       Creates a parse object which parses ISO format
    //              dates (our default).
    // note         This is separate from the above to avoid the
    //              need to parse date formats in the default case.

    /*
     * Simple persistence.
     */
friend APT_Archive &operator ||(APT_Archive &ar, ParseObject &o);
```

```
    APT_Date parse(const char *dateString) const;
    // effect       Parses the date according to the format given to the
    //              constructor.  Returns a date with isValid() == false
    //              if the parse fails.  A possible future optimization
    //              might be to provide a fast parse routine that didn't
    //              do the error checking implied in this guarantee.
    // requires     dateString must be at least as long as the format
    //              string specifies.

    static int lookupMonthString(const char *nptr);
    // effect       Turns a 3 character month string, such as Jan, Feb, etc.
    //              into the equivelent month integer based on a case
    //              insensitive match. It will return -1 if nptr is not a
    //              valid month string.
    // requires     character string must be 3 characters long.

    void set(const char *format);
    // effect               Sets the parse object to describe the given format.
    // requires     Format must be a valid format string.

    APT_String format() const;
    // effect       Return the format corresponding to the parseObject

    APT_String toString(const APT_Date &date) const;
    // effect       Returns a string representing the date specified in the
    //              format given to the constructor of this object.  An
    //              invalid date will be represented by a string of '*'s

    /*
     * String creaters optimized for efficiency.
     */
    APT_UInt32 printSize(const APT_Date &date) const;
    // effect       Returns the buffer size the specified date will
    //              require.

    APT_UInt32 maxPrintSize() const;
    // effect       Returns the maximum size that any date may take using
    //              the format represented by this object.

    bool isPrintFixedSize() const;
    // effect       Returns true if the format represented by this object
    //              will always result in a fixed sized print
    //              representation (i.e. the result of printSize() will be
    //              independent of the argument, and that result will be
    //              equal to maxPrintSize()).

    void printToBuf(const APT_Date &date, char *dest, int *length) const;

    // effect       Writes a printable representation of the date into
    //              the buffer according to the format represented
    //              by this object.  If length is non-null, the
    //              amount of space used will be written to the
    //              location to which it points.  No null will be
    //              written at the end of the representation.
```

```
        //              If an invalid date is passed, the buffer will be
        //              filled with '*'s.
        // requires     dest must point to a buffer that is at least
        //              printSize() long.

        static bool isFormatValid(const char *format);
        // effect       Returns true if format points to a valid format, false
        //              otherwise.

        // Didn't implement () since there's no reason to prefer
        // Date->String or String->Date and I feel a little uncomfortable
        // about overloading.

        // Default destructor, copy constructor, and assignment operator
        // ok.

        bool dateFullySpecified() const;
      private:

        static bool parseFormat(const char *format, ParseObject *results = 0);
        // effect       Parses the specified format returning true if it is a
        //              valid format and false if not.  If the optional
        //              results argument is non-null, it is filled in with the
        //              results of parsing the format.

        bool                    fullYear_;
        bool                    dayAndYear_;

        bool                    monthAs3Char_;
       // This will be true if the month portion of the date format string
       // has been set to %mmm.

        int                     yearOffset_;
        int                     monthOffset_;
        int                     dayOffset_;
        int                     yearCutoff_;
        int                     stringLength_;
        APT_String      startingString_;

        static APT_String       monthArray[12];
        // An array of 3 character abbreviations for each month in the year.
    };

friend class APT_Date::ParseObject;
friend class APT_TimeStamp;

  private:
    static int defaultYearCutoff;

    APT_UInt32  days_;
};

// Other arithmetic functions on the date class; all may be
// implemented in terms of functions described above.
```

```
// Named temporary to work around bug in the aix compiler.
inline APT_Date operator+(const APT_Date &d, APT_Int32 incr)
{
    return APT_Date(d) += incr;
}

inline APT_Date operator+(APT_Int32 incr, const APT_Date &d)
{
    return APT_Date(d) += incr;
}

inline APT_Date operator-(const APT_Date &d, APT_Int32 decr)
{
    return APT_Date(d) -= decr;
}

inline bool operator==(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) == 0);
}

inline bool operator!=(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) != 0);
}

inline bool operator<(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) < 0);
}

inline bool operator<=(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) <= 0);
}

inline bool operator>(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) > 0);
}

inline bool operator>=(const APT_Date &op1, const APT_Date &op2)
{
    return (APT_Date::compare(op1, op2) >= 0);
}

// Support for output to stream.
ostream& operator<< (ostream&, const APT_Date&);

// Persistence
APT_DIRECTIONAL_SERIALIZATION(APT_Date);
APT_DIRECTIONAL_SERIALIZATION(APT_Date::ParseObject);
```

```
#endif // APT_DATE_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1998 Torrent Systems, Inc.  All rights reserved.

/*
Database Interface for serverization. There are two
classes defined here.

1. APT_DataBaseDriver: a simple generic ODBC driver

2. APT_DataBaseSource: an abstraction for querying  and inserting
into the database. Currently does not support any positional updates, as
we do not need this functionality for serverization.

*/

#ifndef APT_DB_SOURCE_H
#define APT_DB_SOURCE_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif
#include <apt_util/odbc.h>          // ODBC type definitions

/* Error call back struture: Contains:

   1. callBack - Function to be called if an error occurs. The
      function takes two arguments - an opaque pointer and an error
      message which will be filled in by the db layer.

   2. context - an opaque object which will be passed back as the
      first argument of the callback function.

   3. An integer value in order to use the errorlogs
*/

typedef struct {
  int (*callBack)(void *, const char *, int i);
  void * context;
} APT_ErrorCallBack;

class APT_DataBaseDriverRep;

class APT_DataBaseDriver {

  public:

  APT_DataBaseDriver();

  virtual ~APT_DataBaseDriver();

  virtual bool connect(const char * networkName,
                       const char * user,
                       const char * passwd,
                       APT_ErrorCallBack *ep = NULL);
```

```
  /*
     Effect: connects to the specified database. The transaction
     isolation level is set to serializable and auto-commit is turned
     off.

     Returns: true if connection succeeded, false otherwise

  */

  const char * networkName();

  const char * userName();

  const char * passWord();

  virtual bool disconnect();
  /*
     Effect: disconnects from the database and frees resources
             associatied with the connection.

     Returns: true if connection succeeded, false otherwise.
  */

  virtual bool execSql(const char * sqlStatement,
                       APT_ErrorCallBack * ep = NULL);
  /*
     Effect: Executes the specified sql statement. This statement
             cannot be one which returns any value other than
             success or failure.
             To execute queries use APT_DataBaseSource

     Returns: true if statement was executed successfully and
              false otherwise.
  */

  virtual bool commit(APT_ErrorCallBack * ep = NULL);
  /*
     Effect: Commits any changes (via execSql above)
     Returns: true if commit was successful.
  */

  virtual bool rollBack(APT_ErrorCallBack * ep = NULL);
  /*
     Effect: rolls back changes up to the last commit point.

     Returns: true if rollback was successful.
  */

  const char * driverErrorMsg();
  /*
     Effect: returns a non-null string if the last driver operation resulted in an
error.
  */

  const char * sqlErrorMsg( HSTMT hstmt = SQL_NULL_HSTMT);
```

```
  /*
    Effect:  returns a non-null string if the last ODBC call resulted in an error.

    Note:   This is a direct interface to the SQLError() ODBC call and
            should be used only if ODBC is being called directly.
            For example, this method is currently used by the APT_DataBaseSource
class to
            get statement related error messages.

            If ODBC is not being called directly, then driverErrorMsg() should be
            used.
  */

  const char * sqlState(HSTMT hstmt = SQL_NULL_HSTMT);
  /*
     Effect: returns the SQLSTATE variable value for the last ODBC call.

     Note: This should only be used if ODBC is being called directly.
           The SQLSTATE string is returned as part of the driverErrorMsg().
  */

  bool uniqueId(int blockSize, int& first);
  bool uniqueId(const char *sequence, int blockSize, int& first);
  /*
     Effect: returns an interval [first,first+blockSize-1] of unique
             ids from the database.

     Returns: true if the interval is valid.
  */

  bool tableExists(const char * tableName);
  /*
    Effect: Returns true if the table exists and is accessible by the current user.
  */

  HENV envHandle();

  HDBC connHandle();

  // Does not transfer ownership
  APT_ErrorCallBack* errorCallBack() const
  { return ep_; }

  private:
  APT_DataBaseDriver(const APT_DataBaseDriver&);                // prevent copying
  APT_ErrorCallBack * ep_;
  APT_DataBaseDriverRep * rep_;
};

class APT_DBColumnDescriptor;
class APT_DataBaseSourceRep;



class APT_DataBaseSource {
```

```
public:

enum DbAccessMode { read, append };
enum ParamType { eDefault=0, eSelectColumn, eInput, eOutput, eInputOutput};
enum Status    { eEof = 0, eOk, eError };

APT_DataBaseSource(APT_DataBaseDriver * const pDataBase);
/*
  Effect: Creates the source object.
  Requires: pDataBase is not null.
*/

virtual ~APT_DataBaseSource();

virtual bool open(const char * SqlQuery,
                  APT_DataBaseSource::DbAccessMode openMode,
                  APT_ErrorCallBack * ep = NULL);

/*
   Effect: Prepares the supplied query.

   Requires: SqlQuery is non-null.
*/

virtual bool openTable(const char * tableName,
                       APT_DataBaseSource::DbAccessMode openMode,
                       const char * selectList = NULL,
                       const char * filter = NULL,
                       APT_ErrorCallBack * ep = NULL);
/*
   Effect: constructs and prepares the query. Does not
     provide any added functionality over open(). The main reason to use
     this form rather than open() is to have better error-checking and
     named field access in append mode.

   Requires: tableName is non-null.
*/

virtual bool reExecute(APT_ErrorCallBack * ep = NULL);
/*
   Effect: re-executes the query in read mode.

   Requires: in read mode and all the parameter binding has been done.
*/

DbAccessMode mode();

virtual bool close(bool commit = false, APT_ErrorCallBack * ep = NULL);
/*
   Effect: flushes any handles, objects that have
           been accumulated. Optionally performs a commit.
*/
```

```
  virtual const char * sqlQuery();
  /*
     Effect: returns the sql query specified in the
             open statment.
  */

  virtual const APT_DBColumnDescriptor * columnDescriptor(int nindex,
                                                          APT_ErrorCallBack * ep =
NULL);

  virtual const APT_DBColumnDescriptor * columnDescriptor(const char * fieldName,
                                                          APT_ErrorCallBack * ep =
NULL);
  /*
     Effect : returns the column description for the nindex'th (named) column

     Returns: If the index or name is out of range, null is returned.
  */

  virtual APT_DataBaseSource::Status getNext(APT_ErrorCallBack * ep = NULL);
  /*
     Effect: fetches the next record.

     Returns: true if eOk if eof has not been reached and
              eEof if it has and eError if an error occurred during the fetch.

     Requires: executing in read mode.

     Example:

     APT_DataBaseDriver *dbdriver = new APT_DataBaseDriver();
     dbdriver->open("tcp metheny 1313", "blair", "blair");
     APT_DataBaseSource  * dbsource = new APT_DataBaseSource(dbdriver);
     dbsource->open("select * from JobInstance where user = 'rlk'",
                    APT_DataBaseSource::read);
     int id;
     int descriptor;
     int user;
     int state;
     APT_TimeStamp * lastStateChange = new APT_TimeStamp();

     dbsource->bind(&id, "id");
     dbsource->bind(&descriptor, "descriptor");
     dbsource->bind(&user, "user");
     dbsource->bind(state, "state");
     dbsource->bind(lastStateChange, "last_state_change");

     while(dbsource->getNext()== APT_DataBaseSource::eOk) {
        cout << id << descriptor << state << *lastStateChange;
        .....
     }

     dbsource()->close();

  */
```

```
virtual bool insertNew(APT_ErrorCallBack *ep = NULL);
/*
   Effect: Inserts a new record.

   Returns: true if insertion was successful.

   Requires: executing in append mode.
*/

virtual int fieldCount(APT_ErrorCallBack *ep = NULL);
/*
   Effect: returns the number of fields per record.

*/

virtual bool bindByIndex (int * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (double * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (float * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (char * string,
                          int length,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (TIME_STRUCT * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (DATE_STRUCT * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

virtual bool bindByIndex (TIMESTAMP_STRUCT * binding,
                          short nindex,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

/*
   Effect: binds the specified storage to the nindex'ed column.
   nindex is zero-based.
```

```
     Returns : true if the binding was possible. The routine
               checks the type of the column to make sure
               the types are compatible whenever possible.

               The pType argument specifies whether the binding is an in, out,
               or in/out parameter or whether it is an output column of a select.
               The default specification is to use eInput for append mode and
               eSelectColumn for read mode.

     Requires:  0 <= nindex < getFieldCount().
  */

  virtual bool bindByName (int * binding, const char * fieldName, APT_ErrorCallBack
*ep = NULL);
  virtual bool bindByName (double * binding, const char * fieldName,
APT_ErrorCallBack *ep = NULL);
  virtual bool bindByName (float * binding, const char * fieldName, APT_ErrorCallBack
*ep = NULL);
  virtual bool bindByName (char * string, int length, const char * fieldName,
APT_ErrorCallBack *ep = NULL);
  virtual bool bindByName (TIME_STRUCT * binding, const char * fieldName,
APT_ErrorCallBack *ep = NULL);
  virtual bool bindByName (DATE_STRUCT * binding, const char * fieldName,
APT_ErrorCallBack *ep = NULL);
  virtual bool bindByName (TIMESTAMP_STRUCT * binding, const char * fieldName,
APT_ErrorCallBack *ep = NULL);

  /*
     Effect: binds the specified storaget to the named column.

     Returns : true if the binding was possible. The routine
               checks the type of the column to make sure
               the types are compatible whenever possible.

     Requires: 1. The source was opened in read mode or was opened via
openTable().i.e.
               These functions cannot be used if opened via open() method in append
mode as
               there is no way to get the column names without parsing SQL.
               2. fieldName is valid.
  */

  virtual bool bindString(char * string,
                          int length,
                          short nindex,
                          short int sqlType,
                          unsigned int precision = 0,
                          short int scale = 0,
                          APT_DataBaseSource::ParamType pType = eDefault,
                          APT_ErrorCallBack *ep = NULL);

  /* Effect: binds the storage pointed to by the string to the nindex'th column.
             This allows binding of text-based input data. Precision and scale only
need
```

```
            to be filled for decimal data.

     Returns: true if the binding was possible, though the checking here is much
              less complete than the in bindByName or bindByIndex.
  */

  bool setNull(short nIndex, APT_ErrorCallBack *ep = NULL);
  bool setNull(const char * fieldName, APT_ErrorCallBack *ep = NULL);
  /*
     Effect: Set the specified field to null. Index is zero-based.
     Returns: false if called in append mod or the index is out of range
  */

  bool isNull(short nIndex, APT_ErrorCallBack *ep = NULL);
  bool isNull(const char * fieldName, APT_ErrorCallBack *ep = NULL);
  /*
     Effect: returns true if the specified field contains a null value.
     Returns : false if called in read mode or index is out of range
  */

  int length(short nIndex, APT_ErrorCallBack *ep = NULL);
  int length(const char * fieldName, APT_ErrorCallBack *ep = NULL);
  /*
     Effect: returns the length of the field (Useful for variable
             checking the length of a varchar before calling
             getNext().

     Returns: the length of the field or -1 if the length cannot
              be determined.

     Requires: can only be called in read mode.
   */

  const char * errorMsg();
  /*
     Effect: returns a non-null string if the last APT_DataBaseSource
             method returned false.

     Note: Do NOT use the underlying dbdriver's driverErrorMsg() method, as it
           will return null strings for any SQL statement related failures.
  */

  APT_DataBaseDriver *driver() const { return pDbDriver_; }

  private:
  APT_DataBaseSource(const APT_DataBaseSource &);              // prevent copying
  // ??? Duplicated in the Solid rep_
  APT_DataBaseDriver * const pDbDriver_;
  APT_DataBaseSourceRep * rep_;

};

// information returned by the getColumnDescriptor() call above.
class APT_DBColumnDescriptor {
  public:
```

```
  unsigned char  *colName;
  short  colNameLen;
  short  colType;
  bool   nullable;
  unsigned long   colLen;
  short  scale;
};

#endif
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_DECIMAL_H
#define APT_DECIMAL_H

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_String;

/* The APT_Decimal class represents a decimal number, such as occur
   in RDBMS systems and COBOL data files.

   Data Representation

   The internal data representation is IBM packed decimal format,
   characterized by precision (number of decimal digits) greater than 1;
   and scale (fixed position of decimal point) between 0 and the precision.
   A decimal with a scale of 0 can represent only integral values.

   APT_Decimal values are always signed.  The number of bytes occupied
   by the data representation (not the class representation) is (P/2)+1.
   This corresponds to a nibble for each decimal digit, followed by a
   sign nibble.  If the number of decimal digits (P) is even, then an
   extra leading nibble (value=0) is inserted to bring the total
   number of nibbles to an even value.

   The sign nybble has the following possible values:
          0xa  +                                Accept on input
       0xb     -                                AOI
       0xc     +       (Preferred)              AOI/Generate on output
       0xd     -       (Preferred)              AOI/Generate on output
       0xe     +                                AOI
       0xf     +       (Also used for unsigned)AOI

   Constructors
```

The APT_Decimal class has a constructor taking a precision and optional scale argument. There is no default constructor. The copy constructor initializes the APT_Decimal to be the same precision, scale, and value of the argument. Note that the decimal assignment operator does not copy the argument's precision and scale; see below.

Numeric Assignment

A decimal may have integer and floating point numeric values assigned to it. Range checking is performed and a requirement failure occurs if the assigned value does not fit within the decimal's available range based on the precision and scale (10^(P-S)).

"Checked" assign member functions are provided to conditionally perform the assignment, returning an error flag if there was a range problem.

Numeric Conversion

The APT_Decimal class provides member functions for converting to integer and floating point values. The asInteger() conversion routine offers several rounding modes, described in an enum declaration below.

Arithmetic

No arithmetic operations are provided. Instead, the client is expected to use numeric conversion and assignment operations to perform arithmetic using native integer and floating point calculations.

An exception is that equality and comparison functions are provided, to facilitate implementation of the corresponding generic functions. A hash function also provided.

If there is strong customer demand, we can implement true decimal arithmetic later. An interim solution was suggested, where APT_Decimal would implement arithmetic operations by internally converting to dfloat and back; this was rejected in favor of requiring the client to manage the costly conversion operations, rather than hiding them behind deceptively simple arithmetic operators.

String Assignment and Conversion

Functions providing assignment from (and conversion to) a string representation are implemented. The string assignment function performs range checking; a "checked" variant is provided to allow a

```
  range check failure to be detected and dealt with.

  Input string formats accepted are of the form:

    [+/-]ddd[.ddd]

  Currently only a single, fixed-width, output string format is
  supported:

    { /-}ddd.[ddd]

  A leading space or minus sign is printed, followed by P-S digits, a
  decimal point, and then S digits.  Leading and trailing zeros are
  not suppressed.  A leading + is accepted on input instead of a
  leading space.  Other string formats may be supported in the
  future; for now other string conversions can be efficiently
  implemented by using the content() member function to access the
  packed decimal representation.

  Decimal Assignment

  When an APT_Decimal value is assigned to another APT_Decimal, the
  left-side's precision and scale are not modified; only the
  right-side's value is copied.  Range checking (with a "checked"
  variant of the assignment function) and rounding (under enum
  control) are implemented as appropriate for assignment of
  APT_Decimal instances with dissimilar precision and/or scale.

  Type System Support

  A raft of member functions are provided to facilitate the
  implementation of various Orchestrate type system items such as
  adapter conversions, import/export generic functions, tsort key
  preparation generic functions, etc.  These are described elsewhere
  (doc/decimal_type.txt).

  Representation Checks

  Several functions have the possibility of returning range/rep check
  failure in a passed arguments; these functions are noted by their
  taking an "APT_Status *checkResult" argument.  These functions only
  return a representation check failure if they notice a
  representation problem as part of their operation; digits that do
  not need to be checked (example: fractional digits when converting
  to integer with rounding mode eTruncZero) may be invalid even with
  an APT_Status return from these functions.  If you need to know if
  the entire representation is ok, call isValid().
*/

class ostream;
```

```
class APT_Decimal
{
public:
  APT_Decimal(int precision, int scale);
  /*
     effect    Constructs this APT_Decimal instance with the indicated
               precision and scale, and a value of zero.
     note      Once constructed, this decimal's precision and scale
               cannot be modified.
     requires  1 <= precision <= 255
               0 <= scale <= precision
  */
  APT_Decimal(const APT_Decimal& rhs);
  /*
     effect    Copies the argument's precision, scale, and value.
     note      Once constructed, this decimal's precision and scale
               cannot be modified.
               If rhs.isValid() is false, then this->isValid()
               becomes false.  (Validity if eZeroRepLegal specified is
               also propagated).
  */

  ~APT_Decimal();

  int precision() const;
  int scale() const;
  /*
     effect    Returns the precision/scale with which this decimal was
               constructed.
  */

  char* content();
  const char* content() const;
  /*
     effect    Returns a pointer to this decimal field's IBM packed
               decimal representation.
     note      The caller must not retain the returned pointer, as the
               underlying storage can be invalidated by many things,
               including (but not least) non-const operations on this
               decimal field and getRecord() or putRecord() calls.
  */

  enum ZeroRep { eZeroRepIllegal, eZeroRepLegal };
  /* member functions that look at a decimal's representation offer
     an option to treat an all-zero rep (not a legal packed decimal
     value) as a valid zero. */

  bool isValid(ZeroRep z=eZeroRepIllegal) const;
  /*
     effect    Tells if this decimal's data representation is valid.
     note      This decimal's data representation can become invalid
```

```
            only via a "packed, nocheck" import operation, where the
            "nocheck" property causes import data integrity checks to
            be bypassed, or via direct writes through content().
            Whether an all-zero representation (such as a "nocheck"
            import might produce) is considered to be a valid rep by
            this function is controlled by the ZeroRep argument.
            Other problems, such as invalid decimal nibbles, or a
            bogus sign nibble, are flagged as an invalid rep.
*/

/**** numeric/decimal assignment ****/

APT_Decimal& operator= (APT_DFloat);
// We don't have an operator=(APT_Int32) as that would result in
// ambiguities with the above.
APT_Decimal& operator= (const APT_Decimal& dec);
/*
   effect    Copies the numeric value into this decimal instance.
   note      For the floating point and decimal arguments, default
             rounding (eTruncZero) is applied.
   requires  The argument value must fit within this decimal's
             precision/scale.
             The dfloat must not be a nan, inf, etc.
             dec.isValid(eZeroRepIllegal) must be true.
*/

enum RoundMode
{
  eTruncZero,                /* Discard material to the right of
                                the surviving digit, regardless of
                                sign.  This corresponds to the
                                COBOL INTEGER-PART function.
                                This is the default rounding mode.
                                Examples: 1.6 -> 1, -1.6 -> -1
                             */
  eRoundInf,                 /* Round towards nearest representable
                                value, breaking ties by rounding
                                towards plus infinity (positive) or
                                minus infinity (negative).  This
                                corresponds to the COBOL ROUNDED
                                phrase.
                                Examples: 1.4 -> 1,   1.5 -> 2,
                                         -1.4 -> -1, -1.5 -> -2
                             */
  eFloor,                    /* truncate towards minus infinity.
                                This corresponds to the IEEE 754
                                Round Down mode.
                                Examples: 1.6 -> 1, -1.4 -> -2
                             */
  eCeil                      /* truncate towards positive infinity.
                                This corresponds to the IEEE 754
```

```
                                    Round Up mode.
                                    Examples: 1.4 -> 2, -1.6 -> -1
                                */
  // not implemented: IEEE 754 Round to Nearest mode (eRoundEven)
};

void assignFromInt32(APT_Int32,
                     APT_Status* checkResult=0);
void assignFromSInt64(APT_Int64,
                     APT_Status* checkResult=0);
void assignFromUInt64(APT_UInt64,
                     APT_Status* checkResult=0);
void assignFromDFloat(APT_DFloat, RoundMode r=eTruncZero,
                      APT_Status* checkResult=0);
void assignFromDecimal(const APT_Decimal&, RoundMode r=eTruncZero,
                       ZeroRep z=eZeroRepIllegal,
                       APT_Status* checkResult=0);
/*
   effect    Copies the numeric value into this decimal instance.
             A range/rep check failure, if noted,  is returned in
             *checkResult, if non-null.  If range/rep check occurs,
             the resulting decimal is undefined.  If a range/rep
             check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   requires  The dfloat must not be a nan, inf, etc.
             If checkResult is null, then no range check failure
             must occur, and (if assigning from an APT_Decimal) the
             argument decimal value must be isValid(z).
*/

/**** numeric conversions ****/

APT_Int32 asInteger(RoundMode r=eTruncZero,
                    ZeroRep z=eZeroRepIllegal,
                    APT_Status* checkResult=0) const;

APT_Int64 asIntegerS64(RoundMode r=eTruncZero,
                    ZeroRep z=eZeroRepIllegal,
                    APT_Status* checkResult=0) const;

APT_UInt64 asIntegerU64(RoundMode r=eTruncZero,
                    ZeroRep z=eZeroRepIllegal,
                    APT_Status* checkResult=0) const;
/*
   effect    Converts this decimal to either a 32 bit integer (if
             asInteger() is called), a 64 bit integer (if asIntegerS64()
             is called), or an unsigned 64 bit integer (if
             asIntegerU64() is called). A range/rep check failure, if
             noted, is returned in *checkResult, if it is non-null. If
             a range/rep check does not occur and checkResult
             was provided, *checkResult is set to APT_StatusOk.
```

```
   requires  If checkResult is null, then this decimal value must be
             isValid(z) and must fit into an 32, 64, or unsigned 64
             bit integer.
*/

APT_DFloat asDFloat(ZeroRep z=eZeroRepIllegal,
                    APT_Status* checkResult=0) const;
/*
   effect    Converts this decimal to a dfloat.  Range/rep check failure
             is returned in *checkResult, if non-null.  If this
             failure occurs, the returned value is undefined.  If
             a range/rep check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   note      No specific rounding mode is applied when computing the
             dfloat value.  The resulting dfloat value might be an
             inexact approximation to the original decimal value.
   requires  If checkResult is null, then this decimal value must be
             isValid(z).
*/

/**** string conversion ****/

void assignFromString(const char* str,
                      int len = -1,
                      RoundMode r=eTruncZero,
                      APT_Status* checkResult=0);
/*
   effect    Interprets the string as a decimal value and sets this
             decimal's value with the result.  A range/rep check
             failure, if noted,  is returned in *checkResult, if
             non-null.  In this case, the resulting decimal is undefined.
             Strings of the following form are accepted:
               [+/-]ddd[.ddd]
             Leading and trailing whitespace is ignored.
             The len argument specifies the length of the string being
             assigned from; if it is -1, the string is assumed null
             terminated.  If a range/rep
             check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   requires  str non-null
             If checkResult is null, then the string must be
             properly formatted and be within this decimal's range.
*/

int stringLength() const;
/*
   effect    Returns precision()+2 (the extra chars are for the
             leading space/minus and the decimal point).
   note      For a given decimal instance, the return value of this
             function is independent of the decimal value.
*/
```

```
APT_String asString(ZeroRep z=eZeroRepIllegal,
                          APT_Status* checkResult=0) const;
void asString(char* buf,
              ZeroRep z=eZeroRepIllegal,
              APT_Status* checkResult=0) const;
void asString(APT_String& str,
              ZeroRep z=eZeroRepIllegal,
              APT_Status* checkResult=0) const;
/*
   effect    Converts this decimal to string form.  A leading space
             or minus sign is printed, and leading/trailing zeros
             are not suppressed.  Trailing spaces will be added in the
             case of a fixed length string field; in the
             APT_String case, the trailing character will be
             str.padChar().
             A range/rep check failure, if noted,  is returned in
             *checkResult, if non-null.  Range failure can occur in the
             APT_String cases for fixed-length string fields if
             stringLength() > the maximum length of the string field.
             In this case, the value of the string written is
             undefined, though we will not write beyond buf +
             stringLength().   If a range/rep
             check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   note      -0 is printed the same as positive 0
   requires  buf non-null
             buf must be allocated to stringLength() bytes (a null
             termination is *not* written).
             If checkResult is null, then this decimal value must be
             isValid(z)
*/

void clear();
// effect     Clears the decimal value (sets it to zero) in an
//            efficient fashion.

/**** equality, comparision, and hashing ****/

// note: -0 and 0 are considered to be equal

friend bool operator== (const APT_Decimal&, const APT_Decimal&);
friend bool operator!= (const APT_Decimal&, const APT_Decimal&);
/* requires: isValid(eZeroRepIllegal) true */

friend bool operator< (const APT_Decimal&, const APT_Decimal&);
friend bool operator<= (const APT_Decimal&, const APT_Decimal&);
friend bool operator> (const APT_Decimal&, const APT_Decimal&);
friend bool operator>= (const APT_Decimal&, const APT_Decimal&);
/* requires: isValid(eZeroRepIllegal) true */
```

```
static int compare(const APT_Decimal& d1, const APT_Decimal& d2,
                   ZeroRep z=eZeroRepIllegal, APT_Status* checkResult=0);
/*
   effect    Returns -1, 0, or +1 according to whether d1 is less
                 than, equal to, or greater than d2.
             Rep check failure is returned in *checkResult, if
             non-null; the value returned by the function in this
             case is undefined.  compare(-0, +0) == 0.  The validity of
             the decimal representations are only checked for those digits
             required to compute the comparison.  If a range/rep
             check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   requires  If checkResult is null, then both decimal values must be
             isValid(z).

*/

APT_UInt32 hash(ZeroRep z=eZeroRepIllegal, APT_Status* checkResult=0) const;
/*
   effect    Computes a hash value for this decimal.
             A range/rep check failure, if noted,  is returned in
             *checkResult, if non-null.  If failure occurs, the
             value returned is undefined.  The validity of the
             decimal representations are only checked for those
             digits required to compute the hash.
             Note that decimals of different precision and scale are
             not guaranteed to hash to the same value.  Decimals of the
             same precision and scale with values +0 and -0 will
             hash to the same value.  If a range/rep
             check does not occur and checkResult was provided,
             *checkResult is set to APT_StatusOk.
   requires  If checkResult is null, then this decimal value must be
             isValid(z)
*/

void setScale(int scale);
/*
   effect    changes the scale of the decimal.
   requires  scale < precision_
*/

// for using APT_Decimal in conditional contexts
operator const void* () const; // non-zero?
bool operator! () const;       // zero?
// requires: isValid(eZeroRepIllegal) true

/**** type system support ****/

/* adapter conversions to/from int32, dfloat, and decimal are
   implemented using assignment/conversion functions listed
   above. */
```

```
  /* import export support: routines to copy to/from various and
     sundry external representations including IBM packed decimal
     (above), zoned decimal, unsigned decimal, separate sign decimal
     (leading/trailing), overpunched sign decimal (leading/trailing),
     edited numeric, and Informix binary pload/punload.  See
     decimal_type.txt */

  // TBD

  /* tsort key preparation: routines to copy to/from a fixed-length
     memcmp()-friendly compatible external representation. */

  // TBD

  friend APT_Archive& operator|| (APT_Archive&, APT_Decimal&);

private:
  // rep is for Orchestrate base/offset protocol
  const unsigned char* rep() const { return (unsigned char *) (*basePtr_
                                                               + offset_); }
  unsigned char* rep() { return (unsigned char *) (*basePtr_ + offset_); }

  // Functions for the type descriptor.
friend class APT_DecimalDescriptor;
  APT_Decimal(); // Precision 1, scale 0;

  void setParams(int precision, int scale);
  // Note that this changes storage alloc.

  static APT_UInt32 representationSize(int precision, int) {
      return precision / 2 + 1;
  }

  void releaseStorage();
  // effect     Releases the storage allocated by this type; renders
  //            this type unusable until and unless the basepointer
  //            and offset are setup to point at other storage.

  // We want the import export functions to be able to get at these as well.
friend class APT_GFIX_Decimal;

  // Internal helper functions.

  int repSize() const { return representationSize(precision_, scale_); }
  // effect     Returns the total size of the representation.

  int leadingZero() const { return (precision_ + 1) % 2; }
  // effect     Return the existence of a leading zero nybble.  This may
  //            occur if the precision isn't using all the space we
  //            needed to reserve?  Will always return 0 or 1; int because
```

```
//            that makes it easy to add to things.

int effectivePrecision() const { return precision_ + leadingZero(); }
// effect     Returns the effective precision (which is 1 less than
//            double the rep size.

int integerDigits() const { return precision_ - scale_; }
// effect     Returns the number of integer digits that this decimal may
//            hold.  Useful shorthand.

int effectiveIntegerDigits() const { return integerDigits() + leadingZero();}
// effect     Returns the number of integer digits, counting any
//            leading zero.

void makeInvalid() { *rep() = 0xff; }
// effect     Makes the decimal invalid.  0xff is an invalid decimal
//            byte under all circumstances, and writing the first byte
//            never overwrites the decimal (as the precision must be
//            greater than 1).

int signNybble() const { return *(rep() + repSize() - 1) & 0xf; }
// effect     Return the value of the sign nybble.

static bool signOk(unsigned char signNybble) {
    return signNybble > 0x9;
}
// effect     Return whether the passed signNybble is ok (assuming
//            the zero rep is illegal).

static bool negativeP(unsigned char signNybble) {
    return signNybble == 0xb || signNybble == 0xd;
}
// effect     Indicates whether the passed sign nybble indicates a negative
//            value.


int integerSize() const { return repSize() - ((scale_ + 1) / 2); }
// effect     Returns the size of the integer portion of the rep, in
//            bytes. Includes both the initial byte (which may have
//            an unused, zero, nybble), and the final byte (which
//            may be half fractional.

unsigned char *fractionStart() { return rep() + repSize() - scale_/2 - 1; }
const unsigned char *fractionStart() const {
    return rep() + repSize() - scale_/2 - 1;
}
// effect     Returns the start of the fractional portion of the
//            representation, rounded down. (i.e. the first location
//            may include a high integer nybble).

bool overlapP() const { return (scale_ % 2) == 0; }
```

```
  // effect     Returns whether or not the integer and fractional
  //            portions of the representation share a byte.

  APT_UInt8 precision_;
  APT_UInt8 scale_;

  char* const* basePtr_;
  APT_UInt32 offset_;

  char* ourAlloc_;              // If set, points to the base of a
                                // memory segment allocated by this
                                // class within which the
                                // representation falls.  If not set,
                                // we are using memory allocated by
                                // some other entity.
  APT_DECLARE_NEW_AND_DELETE(APT_Decimal);
};
APT_DIRECTIONAL_SERIALIZATION(APT_Decimal);

// Support for output to stream.
ostream& operator<< (ostream&, const APT_Decimal&);

#endif // APT_DECIMAL_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ENDIAN_H
#define APT_ENDIAN_H

class APT_ByteOrder
{
public:
  enum ByteOrder
  {
    eBigEndian=0,               // most signifcant byte comes first
    eLittleEndian               // least significant byte comes first
  };

  static ByteOrder nativeOrder()
  { unsigned short testVal = 0x1020; char* probe = (char*) &testVal;
    return (probe[0] == 0x10) ? eBigEndian : eLittleEndian;
  }
  /*
     effect    Tells what the byte ordering is for the processor on we
               are now executing.
  */

  /* Byte-swap routines.  Pointers must refer to valid, non-overlapping
     storage. */

  // this case is provided for benefit of macro-based code generation
  static inline void swap1(const void* src, void* dest)
    { const char* s = (const char*) src; char* d = (char*) dest;
      d[0] = s[0];
    }

  static inline void swap2(const void* src, void* dest)
    { const char* s = (const char*) src; char* d = (char*) dest;
      d[0] = s[1]; d[1] = s[0];
    }

  static inline void swap4(const void* src, void* dest)
    { const char* s = (const char*) src; char* d = (char*) dest;
      d[0] = s[3]; d[1] = s[2]; d[2] = s[1]; d[3] = s[0];
    }

  static inline void swap8(const void* src, void* dest)
    { const char* s = (const char*) src; char* d = (char*) dest;
      d[0] = s[7]; d[1] = s[6]; d[2] = s[5]; d[3] = s[4];
      d[4] = s[3]; d[5] = s[2]; d[6] = s[1]; d[7] = s[0];
```

```
    }


  // vector forms

  static inline void swap1(const void* src, void* dest, APT_UInt32 veclen)
  { const char* s = (const char*) src; char* d = (char*) dest;
    while (veclen--)
    { d[0] = s[0];
      s += 1;
      d += 1;
    }
  }

  static inline void swap2(const void* src, void* dest, APT_UInt32 veclen)
  { const char* s = (const char*) src; char* d = (char*) dest;
    while (veclen--)
    {
      d[0] = s[1]; d[1] = s[0];
      s += 2;
      d += 2;
    }
  }

  static inline void swap4(const void* src, void* dest, APT_UInt32 veclen)
  { const char* s = (const char*) src; char* d = (char*) dest;
    while (veclen--)
    {
      d[0] = s[3]; d[1] = s[2]; d[2] = s[1]; d[3] = s[0];
      s += 4;
      d += 4;
    }
  }

  static inline void swap8(const void* src, void* dest, APT_UInt32 veclen)
  { const char* s = (const char*) src; char* d = (char*) dest;
    while (veclen--)
    {
      d[0] = s[7]; d[1] = s[6]; d[2] = s[5]; d[3] = s[4];
      d[4] = s[3]; d[5] = s[2]; d[6] = s[1]; d[7] = s[0];
      s += 8;
      d += 8;
    }
  }
};

#endif // APT_ENDIAN_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ENV_FLAG_H
#define APT_ENV_FLAG_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

class APT_EnvironmentFlag
/* Class to encapsulate environment variable handling.  Use like this:

   APT_EnvironmentFlag myFlag("MY_ENV_VAR");
   ...
   if (myFlag.isDefined()) { ... }   // same as: if (myFlag) { ... }
   cout << myFlag.value() << endl;

   APT_EnvironmentFlag objects are typically static (global) variables.

   Note: One should avoid interrogating an APT_EnvironmentFlag object
   during STI.  This is because section leader processes' environments
   are not set up properly until later in their startup.
*/
{
public:
  APT_EnvironmentFlag(const char* varName);
  /*
     effect    At point of construction, performs an environment lookup
               to see if the supplied environment variable is defined.
     requires  varName non-null.
  */

  APT_EnvironmentFlag(const char* varName,
                      const char* messageIfDefined,
                      const char* messageIfNotDefined);
  /*
     effect    At point of construction, performs an environment lookup
               to see if the supplied environment variable is defined.
               The corresponding message is sent to the log depending
               on whether or not the environment variable is defined.
     requires  varName non-null.
  */
```

```
// default copy/assign/dtor OK

bool isDefined() const;
/*
   effect    Tells if an environment variable of the name supplied
             to the constructor is defined (even if it has no value).
*/

operator const void*() const; // same sense as isDefined()
bool operator!() const;       // opposite sense as isDefined()

APT_String valueStr() const;
/*
   effect    Returns the value of the environment variable.
   note      Returns empty string both when the environment variable
             is defined with no value, and when the environment variable
             is not defined.  Use isDefined() to distinguish between
             these cases.
*/

APT_String nameStr() const;
/*
   effect    Returns the name of the environment variable passed
             to the constructor.
*/

void set(const char *value);
/*
   effect    Sets the underlying environment variable to the value
                  specified.  The value is copied into its own storage,
             and the pointer need not be preserved by the caller.
   note      This is *NOT* thread safe!!!
 */

static void updateFlags();
/*
   effect    Causes APT_EnvironmentFlag object's cached values to be
             invalidated.  Such value will be re-read upon the first
             subsequent request.
*/


/* functions for backwards compatibility with APT_UTEnvFlag */

bool on() const;
/*
   effect    Returns true iff the environment variable is defined
```

```
               with a non-empty value.
     note      This is not the same as isDefined().
  */
  bool off() const { return !on(); }

  const char* value() const;
  /*
     effect    Returns environment variable's value.  Returns 0 if
               the environment variable is not defined.
  */

private:
  void initialize() const
  { if (countWhenRead_ != sCountVal) readVar(); }

  void readVar() const;

  int countWhenRead_;
  APT_String varName_;
  bool isDefined_;
  APT_String varValue_;
  char* value_;

  static int sCountVal;
};
typedef APT_EnvironmentFlag APT_UTEnvFlag; // back compatibility

#endif // APT_ENV_FLAG_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ERRIND_H
#define APT_ERRIND_H

// This file contains declarations and definitions related to the
// general error message reporting format, including the module id
// definitions, error index typedefs, and whatever top level error
// information is required.  This file should be changed with care,
// as it is included by much of the orchestrate system.

#ifndef APT_INTS
#include <apt_util/ints.h>
#endif

class APT_String;

// Top level class, for context
class APT_Error
{
  public:

    // Mostly this class is just to get a new type that holds the module Id.
    // In time, the constructor may be setup to confirm that no other
    // object of this class with the same module id has been constructed.
    class SourceModule
      {
        public:
          SourceModule(const char *moduleId);
          // effect     Construct an object with the given module id.  The
          //            module id is a four character string that does
          //            not include nulls.  If a null is encountered
          //            in processing the moduleId, the string is
          //            blank filled as a convenience.
          // requires   No other object has been constructed with this
          //            moduleId.  moduleId points to a character
          //            string (e.g. is not null)


          APT_String getModuleId() const;
          // effect     Returns the moduleId as a string.

          // Default destructor ok.

          // Prohibit copy and assignment
        private:
          SourceModule(const SourceModule &rhs); // Unimplemented
```

```
        SourceModule &operator =(const SourceModule &rhs);
        // Unimplemented.

        // DATA

        char          moduleId_[4];
    };

  // Error Index range object.
  class IndexRange {
    public:
      IndexRange(int start, int end, const char *description,
                 SourceModule &errorModule);
      // effect       Allocate a top-level range of error indices for the
      //              given module.  The "description" argument
      //              should describe the purpose to which the range
      //              is being put.  The errorModule should be the
      //              APT_ErrorSourceModule object within which
      //              these error indices will be used.
      // requires     The range must not have been allocated within this
      //              module.

      IndexRange(int start, int end, const char *description,
                 IndexRange &parent);
      // effect       Construct an error index subrange within an already
      //              existing error index range identified by
      //              "parent".  This ranges goes from start
      //              (inclusive) to end (exclusive).  The
      //              "description" argument should describe the
      //              purpose to which the range is being put.
      // requires     This range must have been allocated to our parent.

      ~IndexRange();

    private:
      // Prohibit copying and assignment.
      IndexRange &operator =(const IndexRange &rhs);
      IndexRange(const IndexRange &range);

    private:
      // Data

      int rangeStart_;
      int rangeEnd_;
      APT_String* description_;
      IndexRange *parent_;
      SourceModule *module_;
  };
```

```
    // Severity of errors.
    enum ErrorSeverity {
      eInfo = 0,        // Minor status updates
      eWarn = 1,        // Possibly bad (user needs to decide)
      eError = 2,       // Definitely bad (we know it)
      eFatal = 3        // We're going down!
      };

    // Define error indices.
    typedef APT_UInt32  ErrorIndex;
    // Define globally visible error indices
#define APT_ERROR_INDEX_UNSPECIFIED     0

};


// Reference to globally visible module ids

// For user-written code such as operators.
extern APT_Error::SourceModule APT_userErrorSourceModule;

// For error messages written to stdout or stderr of an operator.
extern APT_Error::SourceModule APT_subprocErrorSourceModule;

// Temporary; will go away soon.
extern APT_Error::SourceModule APT_tmpErrorSourceModule;

#endif  // APT_ERRIND_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ERRLOG_H
#define APT_ERRLOG_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ERRIND_H
#include <apt_util/errind.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

class ostream;
class ostrstream;

class APT_ErrorLogAccumulator;

class APT_ErrorLog
/* The '*' operator yields an ostream to which information can be
   printed.  The member functions logError() and logWarning()
   accumulate whatever has been printed via '*' into the error and
   warning logs, respectively.

   The error and warning logs can be queried and read out.
*/
{
public:

  APT_ErrorLog(APT_Error::SourceModule &module);
  // effect     Constructs to the empty state, specifying the module in
  //            which errors will be considered to exist.
  //            Both hasError() and hasWarning() will return false.
  // notes      The default constructor will soon dissapear.

  ~APT_ErrorLog();

  void setModuleId(APT_Error::SourceModule &module);
  // effect    Changes this error log's module affiliation
```

```
void setIdent(const char * ident);
// effect    Changes this error log's default ident tag

APT_String ident() const { return defaultIdent_; }
// effect    Retrieve this error log's default ident tag

ostream& log();
ostream& operator* ();
//
// effect     Returns an ostream to which error or warning
//            information can be printed.

void logError(APT_Error::ErrorIndex errorIndex);
void logError(APT_Error::SourceModule &module,
              APT_Error::ErrorIndex errorIndex);
void logError(const APT_String& ident,
              APT_Error::SourceModule &module,
              APT_Error::ErrorIndex errorIndex);
// effect     Takes whatever has been printed to log()'s ostream and
//            appends it to an accumulated error list, and resets
//            log()'s ostream to the empty state.  The errorIndex
//            to be associated with the error is passed as an argument;
//            the error module is taken from the construction.
//            hasError() becomes true.
//            The second version of this call allows the module id to
//            be specified on a per message basis.
// requires   Some error message text has been placed on the log() stream.

bool hasError() const
{ if (!log_) return false;
  else return hasError_();
}
//
// effect     Tells if logError() has been called since the last call
//            to getErrorAndReset().

int errorCount() const
{ if (!log_) return 0;
  else return errorCount_();
}
//
// effect     Returns the number of logError()s since the last call
//            to getErrorAndReset().

APT_String getError(int index, APT_Error::SourceModule **module,
                            APT_Error::ErrorIndex *errorIndex,
                            APT_String *ident) const;
// effect     Returns the error string indexed by INDEX,
```

```
//            along with its ident, module, and error Id.
//            ident, module or errorIndex may be null, in which case this
//            value isn't returned.
// requires   0 <= index < errorCount()
// notes      the module of a particular error may be different from the
//            module specified for this error log if another log has
//            been appended to this one via appendLog.
//            for similar reasons, the ident of a particular error
//            may be different from the ident of this error log.
//            Will warn (possibly assert) if this function is called when
//            there is still material on the main stream.

void resetError();

void logWarning(APT_Error::ErrorIndex errorIndex);
void logWarning(APT_Error::SourceModule &module,
                APT_Error::ErrorIndex errorIndex);
void logWarning(const APT_String& ident,
                APT_Error::SourceModule &module,
                APT_Error::ErrorIndex errorIndex);
// effect     Takes whatever has been printed to log()'s ostream and
//            appends it to an accumulated warning list, and resets
//            log()'s ostream to the empty state.  The errorIndex
//            to be associated with the error is passed as an argument;
//            the error module is taken from the construction.
//            hasWarning() becomes true.
//            The second version of this call allows the module id to
//            be specified on a per message basis.
// requires   Some error message text has been placed on the log() stream.

bool hasWarning() const
{ if (!log_) return false;
  else return hasWarning_();
}
//
// effect     Tells if logWarning() has been called since the last call
//            to getWarningAndReset().

int warningCount() const
{ if (!log_) return 0;
  else return warningCount_();
}
//
// effect     Returns the number of logWarning()s since the last call
//            to getWarningAndReset().

APT_String getWarning(int index, APT_Error::SourceModule **module,
                              APT_Error::ErrorIndex *errorIndex,
                              APT_String *ident) const;
```

```
// effect     Returns the warning string indexed by the warning index
//            described, along with its ident, module, and error Id.
//            ident, module or errorIndex may be null, in which case this
//            value isn't returned.
// requires   0 <= Index < warningCount()
// notes      the module of a particular warning may be different from the
//            module specified for this error log if another log has
//            been appended to this one via appendLog.
//            for similar reasons, the ident of a particular warning
//            may be different from the ident of this error log.
//            Will warn (possibly assert) if this function is called when
//            there is still material on the main stream.

void resetWarning();

void logInfo(APT_Error::ErrorIndex errorIndex);
void logInfo(APT_Error::SourceModule &module,
             APT_Error::ErrorIndex errorIndex);
void logInfo(const APT_String& ident,
             APT_Error::SourceModule &module,
             APT_Error::ErrorIndex errorIndex);
// effect     Takes whatever has been printed to log()'s ostream and
//            appends it to an accumulated info list, and resets
//            log()'s ostream to the empty state.  The errorIndex
//            to be associated with the error is passed as an argument;
//            the error module is taken from the construction.
//            hasInfo() becomes true.
//            The second version of this call allows the module id to
//            be specified on a per message basis.
// requires   Some error message text has been placed on the log() stream.

bool hasInfo() const
{ if (!log_) return false;
  else return hasInfo_();
}
//
// effect     Tells if logInfo() has been called since the last call
//            to getInfoAndReset().

int infoCount() const
{ if (!log_) return 0;
  else return infoCount_();
}
// effect     Returns the number of logInfo()s since the last call
//            to getInfoAndReset().

APT_String getInfo(int index, APT_Error::SourceModule **module,
                           APT_Error::ErrorIndex *errorIndex,
                           APT_String *ident) const;
```

```
 // effect     Returns the info string indexed by the info index
 //            described, along with its ident, module, and error Id.
 //            ident, module or errorIndex may be null, in which case this
 //            value isn't returned.
 // requires   0 <= Index < infoCount()
 // notes      the module of a particular info may be different from the
 //            module specified for this error log if another log has
 //            been appended to this one via appendLog.
 //            for similar reasons, the ident of a particular info
 //            may be different from the ident of this error log.
 //            Will warn (possibly assert) if this function is called when
 //            there is still material on the main stream.

 void resetInfo();

 void appendLog(APT_ErrorLog &sublog) { appendLog(sublog, ""); }
 void appendLog(APT_ErrorLog &sublog, const char *prefix) {
   appendLog(sublog, prefix, prefix, prefix);
 }
 void appendLog(APT_ErrorLog &sublog, const char *errorPrefix,
                const char *warningPrefix, const char *infoPrefix);
 // effect     Appends the contents of the specified log to this one,
 //            resetting it in the process.
 //            errorPrefix, warningPrefix, and infoPrefix, if specified,
 //            indicate the prefix to be put on each message of the given
 //            type during the append.  If the single prefix argument
 //            version is used, that string is prefixed to all types of
 //            messages.
 // notes      This may result in a single log containing multiple
 //            moduleIds.

 void dump() { if (log_) dump(""); }
 void dump(const char *prefix) { dump(prefix, prefix, prefix); }
 void dump(const char *errorPrefix, const char *warningPrefix,
           const char *infoPrefix);
 // effect     Dumps all of the information contained in this errorLog
 //            to the user using APT_DETAIL_LOGMSG, and resets all of
 //            the three accumulators.
 //            errorPrefix, warningPrefix, and infoPrefix, if specified,
 //            indicate the prefix to be put on each message of the given
 //            type during the dump.  If the single prefix argument
 //            version is used, that string is prefixed to all types of
 //            messages.
 //            This function is a no-op if there isn't anything in the
 //            log.

 void prependErrors(const char* prefix);
 void prependWarnings(const char* prefix);
 void prependInfo(const char* prefix);
```

```
  void prepend(const char* prefix) { prependErrors(prefix);
                                     prependWarnings(prefix);
                                     prependInfo(prefix); }
  // effect     TEMPORARY; do not document.  Prepends given string to
  //            all entries of the specified type.

  void reset() { resetError(); resetWarning(); resetInfo(); }

private:
  // prohibit copy/assign
  APT_ErrorLog(const APT_ErrorLog&);
  APT_ErrorLog& operator= (const APT_ErrorLog&);

  // Helper function for lazy allocation.
  void allocateIfNeeded();

  bool hasError_() const;
  bool hasWarning_() const;
  bool hasInfo_() const;

  int errorCount_() const;
  int warningCount_() const;
  int infoCount_() const;

  ostrstream* log_;

  APT_Error::SourceModule *defaultModule_;
  APT_String defaultIdent_;

  APT_ErrorLogAccumulator *errors_;
  APT_ErrorLogAccumulator *warnings_;
  APT_ErrorLogAccumulator *infos_;

  APT_UInt32 seq_;
};

#define APT_DUMP_LOG(log_, body_)                                       \
  do {                                                                  \
    ostrstream apt_dump_log_ostrstream;                                 \
    apt_dump_log_ostrstream << body_ << ends;                           \
    char* apt_dump_log_buffer = apt_dump_log_ostrstream.str();          \
    (log_).dump(apt_dump_log_buffer);                                   \
    delete[] apt_dump_log_buffer;                                       \
  } while (0)

#define APT_APPEND_LOG(log_, other_, body_)                             \
  do {                                                                  \
    ostrstream apt_dump_log_ostrstream;                                 \
    apt_dump_log_ostrstream << body_ << ends;                           \
```

```
    char* apt_dump_log_buffer = apt_dump_log_ostrstream.str();          \
    (log_).appendLog((other_), apt_dump_log_buffer);                    \
    delete[] apt_dump_log_buffer;                                       \
  } while (0)

#define APT_PREPEND_LOG(log_, body_)                                    \
  do {                                                                  \
    ostrstream apt_dump_log_ostrstream;                                 \
    apt_dump_log_ostrstream << body_ << ends;                           \
    char* apt_dump_log_buffer = apt_dump_log_ostrstream.str();          \
    (log_).prepend(apt_dump_log_buffer);                                \
    delete[] apt_dump_log_buffer;                                       \
  } while (0)

#endif // APT_ERRLOG_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_ERRORCONFIG_H
#define APT_ERRORCONFIG_H

#ifndef APT_INTS
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_ERRIND_H
#include <apt_util/errind.h>
#endif

// Note that errorconfig.h cannot be dependent upon archive.h (as archive.h
// depends on the error system).

class APT_Archive;

#include <sys/time.h>

class ostream;

// Objects of class APT_ErrorConfigMapObject hold a set of mappings from
// APT_ErrorConfiguration::SLPlayer to a triplet of (ip addr, node
// name, operator id).
class APT_ErrorConfigMapObject;
class APT_ParseError;
class APT_MessageStore;

// Class that defines the format of error messages, and formats them
// properly.  Note that there is only one object of this type, and it is
// accessible through the static member function
// APT_ErrorConfiguration::get().

// The error configuration object has a default configuration, which
// will be overriden by the environment variable APT_ERROR_CONFIGURATION,
// which may in turn be overridden by calls to the various setter functions
// below (setConfiguration & etc).  This hierarchy is enforced with no action
// required on the user's part--specifically, setEnvironmentalConfiguration
// does not need to be called.  That function is provided as a convenience
// for those that want to change the base defaults but allow the
// APT_ERROR_CONFIGURATION to override that default; in such a case, the
// programmer would:
//      Set the default configuration they wished (possibly
//              via setCommandConfiguration() or setOrchestrateConfiguration())
```

```
//      Call setEnvironmentalConfiguration()

class APT_ErrorConfiguration
{
  public:
    bool setConfiguration(const char *config, APT_ParseError *err = 0);
    // effect   Set the configuration of this error object according to the
    //          string passed.  Returns false if the string was not
    //          parsable.  If the string a not parseable, and a parse error
    //          object was passed in, the reasons for the inability to
    //          parse are returned.  Note that the order of the various
    //          error fields is fixed; the order of the keywords in the
    //          config argument is ignored.
    // requires config != 0

    void setEnvironmentalConfiguration();
    // effect   Sets the configuration of the global error configuration object
    //          according to the environmental variable
    //          APT_ERROR_CONFIGURATION.  Note that by default the error
    //          configuration object will have this configuration, but
    //          there are cases in which it is useful to reset the
    //          configuration according to that variable (such as wanting
    //          to change the default which the environment variable
    //          overrides).  If the environment variable is not set, this
    //          is a nop.

    void setCommandConfiguration();
    void setOrchestrateConfiguration();
    // effect   Set the configuration of the global error object to either the
    //          default command configuration or the default orchestrate
    //          configuration, as specified.  If this is being done to
    //          change the default which may be overridden by the
    //          environment variable, setEnvironmentalConfiguration() should
    //          be called after calling these routines.

    void setJobIdNumber(int jobId) { jobId_ = jobId; }
    // effect   Set the job id used by this error configuration object.
    //          Default jobid is zero.

    int jobIdNumber() const { return jobId_; }
    // effect   Return the job id set for this error configuration object.

    void setSeverity(bool on = true)           { severity_ = on; }
    void setVseverity(bool on = true)          { vseverity_ = on; }
    void setJobId(bool on = true)              { jobid_ = on; }
    void setModuleId(bool on = true)           { moduleid_ = on; }
    void setErrorIndex(bool on = true)         { errorindex_ = on; }
    void setTimeStamp(bool on = true)          { timestamp_ = on; }
    void setIpAddr(bool on = true)             { ipaddr_ = on; }
    void setNodePlayer(bool on = true)         { nodeplayer_ = on; }
    void setNodeName(bool on = true)           { nodename_ = on; }
```

```
    void setOperatorId(bool on = true)          { operatorid_ = on; }
    void setMessage(bool on = true)             { message_ = on; }
    void setLengthPrefix(bool on = true)        { lengthprefix_ = on; }
    // effect   Set the specified configuration option for this
    //          error configuration object.

    bool severity() const                 { return severity_; }
    bool vseverity() const                { return vseverity_; }
    bool jobId() const                    { return jobid_; }
    bool moduleId() const                 { return moduleid_; }
    bool errorIndex() const               { return errorindex_; }
    bool timeStamp() const                { return timestamp_; }
    bool ipAddr() const                   { return ipaddr_; }
    bool nodePlayer() const               { return nodeplayer_; }
    bool nodeName() const                 { return nodename_; }
    bool operatorId() const               { return operatorid_; }
    bool message() const                  { return message_; }
    bool lengthPrefix() const             { return lengthprefix_; }
    // effect   Returns the setting of the specified configuration option.

    static APT_ErrorConfiguration &get();
    // effect   Returns the global error configuration object.

    // ** Publically advertised functions above this line; internal
    // class details below.

    // The following code is responsible for formatting error messages
    // according to the configuration described above.  It needs to
    // work in the standard orchestrate environment (with conductor,
    // section leader, and players).  For that reason, it breaks out
    // the process of outputing an error message as follows:
    //          output is called external to this routine.
    //                  It timestamps the error message and calls
    //                  the router specified for the global
    //                  APT_ErrorConfiguration object.
    //          the router "arranges" to call outputStream on
    //                  the conductor, with the origin of the message
    //                  added.
    //          outputStream consults the configuration of the global
    //                  APT_ErrorConfiguration object, does any
    //                  mapping of the origin argument to other
    //                  possible origin formats, and outputs the message.
    // The default router just turns around and calls output stream
    // directly.  The process manager is responsible for putting a
    // router in place on section leaders and players that arranges
    // for outputStream to be called on the conductor.
    // The process manager is also responsible for (on the conductor)
    // setting the mapping object that defines the mapping used
    // for converting from SLPlayer to the other formats.
    //
    // The process manager portion of this process is class
```

```
// APT_ErrorConfigManagement, in apt_internal/errorroute.h

struct SLPlayer {
    int sectionLeader_;
    int player_;

    friend APT_Archive &operator||(APT_Archive &ar, SLPlayer &slp);
};

struct TimeStamp {
    APT_UInt16      yearsSince1900_;
    APT_UInt8       monthOfYear_;
    APT_UInt8       dayOfMonth_;
    APT_UInt32      secondsSinceMidnight_;
    APT_UInt32      sequenceNumber_;

    friend APT_Archive &operator||(APT_Archive &ar, TimeStamp &ts);
};

void setMapping(const APT_ErrorConfigMapObject &map);
// effect   Sets the mapping to be used by this error configuration
//          object between structures of type
//          APT_ErrorConfigSLPlayer and all of the other
//          originator objects used.
// note     The default map object only has originator "0,0" in
//          it, and maps that node,player id to:
//                  IP addr         127.0.0.1
//                  Nodename        localhost
//                  opid            <default,0>

void outputStream(ostream &ostr,
                  const SLPlayer &origin,
                  const TimeStamp &ts,
                  const char moduleString[4],
                  APT_Error::ErrorSeverity severity,
                  APT_Error::ErrorIndex index,
                  const char *message);
// effect   Output the specified message to the specified stream,
//          formatting it properly using the configuration and map
//          object specified for this object.
// DEPRECATED

void outputPlayerMessage(const SLPlayer &origin,
                  const TimeStamp &ts,
                  const char moduleString[4],
                  APT_Error::ErrorSeverity severity,
                  APT_Error::ErrorIndex index,
                  const char *message,
                  const char *ident);
// effect   Map the origin to an ECMValue and calls the message
//          store for the configuration to output the message.
```

```
   void setMessageStore(APT_MessageStore* store);
   // effect   Replace the default message store.


   APT_MessageStore* getMessageStore() const;
   // effect   Gets the current message store for this configuration.

   // Routing of error messages
   typedef void (*Router)(const TimeStamp &ts,
                          const char moduleString[4],
                          APT_Error::ErrorSeverity severity,
                          APT_Error::ErrorIndex index,
                          const char *message,
                          const char *ident);

   void setRouter(Router router);
   // effect   Set the router to be used by this error config object.
   //          The default router puts a (0,0) node player origin
   //          on the message and hands it to outputStream(cerr).

   static void defaultRouter(const TimeStamp &ts,
                             const char moduleString[4],
                             APT_Error::ErrorSeverity severity,
                             APT_Error::ErrorIndex index,
                             const char *message,
                             const char *ident);
   // effect   Put a (0,0) node player origin on the message and hand
   //          it to outputStream(cerr, ...)

   static void output(const APT_Error::SourceModule &module,
                      APT_Error::ErrorSeverity severity,
                      APT_Error::ErrorIndex index,
                      const char *message,
                      const char *ident);
   // effect   Timestamp the message and call the default router with
   //          the timestamp/message combo.

   // Other output functions will be added as useful to support the
   // various error macros.

   static void getTimeStampValue(TimeStamp *ts, int *seqno, time_t *last);
   // effect   Helper routine which unfortunately needs to be used
   //          by routines outside of this class.  Takes a passed in
   //          TimeStamp object, and fills it in with the appropriate
   //          current time stamp.  seqno and last are in/out arguments;
   //          they are each the sequence number and time_t appropriate to
   //          the last call to this routine, and are updated with the
   //          same values for this call to the routine.
   //
```

```
    // requires ts != 0, seqno != 0, last != 0

  private:
    // Since there can be only one of these objects, we make the
    // constructor private; it will only be used by get().

    APT_ErrorConfiguration();
    // effect   Create an error configuration.  The specification string,
    //          if specified, describes the configuration of the error
    //          output according to the format described in the error
    //          message format design document.
    //          The default specification may be set by the environment
    //          variable APT_ERROR_CONFIGURATION; in the absence of that
    //          environmental variable, it is specified by the initialization
    //          of the APT_ErrorConfiguration::defaultInitString (not
    //          specified here as I expect it to be volatile for a while
    //          yet).

    static const char *defaultInitString_;
    static const char *commandInitString_;
    static const char *orchestrateInitString_;

    APT_MessageStore *messageStore_;

    int jobId_;

    bool severity_;
    bool vseverity_;
    bool jobid_;
    bool moduleid_;
    bool errorindex_;
    bool timestamp_;
    bool ipaddr_;
    bool nodeplayer_;
    bool nodename_;
    bool operatorid_;
    bool message_;
    bool lengthprefix_;

    class APT_ErrorConfigMapObject *    map_;
    Router                              router_;
};

#endif  // APT_ERRORCONFIG_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_EXCEPTION_H
#define APT_EXCEPTION_H

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#ifndef APT_CONDITION_H
#include <apt_util/condition.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

class APT_StackTrace;

/* Exception classes are generally (but not always) declared nested
   within the class to which they apply.  However, exception classes
   should not be nested within a template class, and should not
   themselves be templates.

   To derive an APT exception class:

   1. Derive (directly or indirectly) from APT_Exception.
   2. Use the APT_DECLARE_EXCEPTION and APT_IMPLEMENT_EXCEPTION macros
      for concrete exception classes.  These macros are not used for
      abstract exception base classes.
   3. Use the RTTI macros if checked casts to your exception class are
      desired.  See rtti.h.
   4. Define constructor(s) as appropriate.
   5. Override description_() if the derived exception class contains
      information that can be formatted into a description string.
*/

class APT_Exception
{
  APT_DECLARE_RTTI(APT_Exception);

public:
  virtual ~APT_Exception();
  /* default ctor, copy ctor, assign OK */

  virtual APT_String name() const=0;

  APT_String description() const;
```

```
/*
   effect    Calls description_() and returns a string of the
             following form:
               "<classname>: <description_>"
*/

typedef void (*AbortHook)(const APT_Exception&);

static void abort(const APT_Exception& e);
/*
   effect    Called by APT_THROW if C++ exception handling is not
             supported.
             Calls the abort hook (if any), prints to cerr (or logs,
             TBD) a message, and then aborts program execution via
             ::abort().
   note      When C++ exception handling is supported, then
             APT_THROW will not invoke this function.  However, a
             top-level exception handler might typically abort
             execution via this function.
*/

static AbortHook setAbortHook(AbortHook);
/*
   effect    Specifies the hook function to be called by abort().
             Returns the previous hook.  0 means no hook.  Initially
             there is no hook.
*/

static void throwHook(const APT_Exception&);
/*
   effect    Function called by APT_THROW() macro prior to throwing
             exception (or aborting if there's no exception
             support).
   note      This function is a handy place to set a debugger
             breakpoint.
*/

static void catchInstalled();
/*
   effect    Should be called from within the framework once a catch
             has been setup for exceptions.
             Prior to this (especially, during STI), throwHook()
             will print exception information at the point of
             throw.
             After this function is called, no exception information
             is printed (unless the catch handler chooses to).
   TBD       What about STD?
*/

static void setStackTrace(APT_StackTrace* stackTrace);
```

```
protected:
  virtual APT_String description_() const;
  /*
     effect    Can be overridden to provide a detailed description of
               the exception instance.
               The default implemenentation returns:
                 "No further information provided."
  */

private:
  friend class APT_ExceptionUT;

  static int sCatchInstalled;
  static AbortHook sHook;
  static int sNoAbort;          // for UT
  static APT_StackTrace* stackTrace;
};

#ifdef __EXCEPTIONS__
#  define APT_THROW(e) do { APT_Exception::throwHook(e); throw e; } while (0)
#else
#  define APT_THROW(e) do { APT_Exception::throwHook(e);                \
                            APT_Exception::abort(e); } while (0)
#endif

/* derivation example:

   in list.h:

     class APT_List
     {
     public:
       class Errors : public APT_Exception
       { APT_DECLARE_RTTI(APT_List::Errors); };

       class Empty : public Errors
       { APT_DECLARE_RTTI(APT_List::Empty);
         APT_DECLARE_EXCEPTION(APT_List::Empty);
       };

       class NotIn : public Errors
       { APT_DECLARE_RTTI(APT_List::NotIn);
         APT_DECLARE_EXCEPTION(APT_List::NotIn);
       };

       // list member functions, some of which throw APT_List::Empty or
       //  APT_List::NotIn
       // ...
     };
```

```
   in list.C

     APT_IMPLEMENT_RTTI_ONEBASE(APT_List::Errors, APT_Exception);

     APT_IMPLEMENT_RTTI_ONEBASE(APT_List::Empty, APT_List::Errors);
     APT_IMPLEMENT_EXCEPTION(APT_List::Empty);

     APT_IMPLEMENT_RTTI_ONEBASE(APT_List::NotIn, APT_List::Errors);
     APT_IMPLEMENT_EXCEPTION(APT_List::NotIn);

     // implementation of list member functions...
*/

/* derivation macros */

#define APT_DECLARE_EXCEPTION(T)        \
public:                                 \
  virtual APT_String name() const;      \
private:                                \
  typedef T APT_dummy_typedef_so_semicolon_can_be_used_with_macro

#define APT_IMPLEMENT_EXCEPTION(T)              \
APT_String T::name() const { return #T; }       \
  class APT_dummy_class_decl_so_semicolon_can_be_used_with_macro

#endif // APT_EXCEPTION_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1998 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_FAST_ALLOC_H
#define APT_FAST_ALLOC_H

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#include <stddef.h>
#include <assert.h>
#include <string.h>

// Enables expensive checks for consistency, corruption
// 1 is basic; 2 is nonlinear checks
// #define DEBUG_FAST_ALLOC 2

class APT_FixedSizeAllocator
{
public:
  APT_FixedSizeAllocator(APT_UInt32 eltSize);
  /*
     effect    Constructs an allocator that will allocate elements
               of the indicated size.
  */

  void* alloc();
  /*
     effect    Returns storage to eltSize bytes.  The storage will
               be aligned as 0 (mod 8) plus some multiple of
               eltSize.
     note      As needed, the system allocator is called to obtain a
               chunk of storage that is divided into several
               eltSize-sized parcels.
  */

  void free(void* elt);
  /*
     effect    Places the indicated elt onto this allocator's
               freelist.
               If elt is 0 there is no effect.
     note      free() does not return storage to the global heap.
```

```
    requires  elt must have been returned from this allocator's
              alloc(), and not yet free()d.
  */

private:
  APT_FixedSizeAllocator(const APT_FixedSizeAllocator&);
  APT_FixedSizeAllocator& operator= (const APT_FixedSizeAllocator&);

  void alloc_();

#if defined(DEBUG_FAST_ALLOC)
  void validate();
  unsigned long eltsFree_;              // freeList_ length
  unsigned long eltsOutstanding_;       // alloc() - free() count
#endif

  struct Link
  {
    Link* next_;
  };

  Link* freeList_;
  APT_UInt32 eltSize_;          /* rounded up to be a multiple
                                   of sizeof(Link*) */
  static bool sDisable;         // use regular ::new and ::delete instead
};

/* note: these macros should only be used on concrete classes
   (not base classes) */
#define APT_DECLARE_NEW_AND_DELETE(C)                   \
public:                                                 \
  void* operator new(size_t)                            \
  { return sClassAllocator().alloc(); }                 \
  void operator delete(void* elt)                       \
  { sClassAllocator().free(elt); }                      \
private:                                                \
  static APT_FixedSizeAllocator* sAllocObj;             \
  static APT_FixedSizeAllocator& sClassAllocator()      \
  { if (!sAllocObj) initAllocObj();                     \
    return *sAllocObj;                                  \
  }                                                     \
  static void initAllocObj()

#define APT_IMPLEMENT_NEW_AND_DELETE(C)                 \
  void C::initAllocObj()                                \
  { sAllocObj = new APT_FixedSizeAllocator(sizeof(C)); }\
  APT_FixedSizeAllocator* C::sAllocObj
```

```
/* Macro versions for two-arg templates; usage example:
     APT_DECLARE_NEW_AND_DELETE_2(RWTValAssocLink<APT_FieldSelector,int>);
   The plain macro cannot be used because of the comma within the
   template specification.
*/
#define APT_DECLARE_NEW_AND_DELETE_2(TEMPL,T1,T2)       \
public:                                                 \
  void* operator new(size_t)                            \
  { return sClassAllocator().alloc(); }                 \
  void operator delete(void* elt)                       \
  { sClassAllocator().free(elt); }                      \
private:                                                \
  static APT_FixedSizeAllocator* sAllocObj;             \
  static APT_FixedSizeAllocator& sClassAllocator()      \
  { if (!sAllocObj) initAllocObj();                     \
    return *sAllocObj;                                  \
  }                                                     \
  static void initAllocObj()

#define APT_IMPLEMENT_NEW_AND_DELETE_2(TEMPL,T1,T2)     \
  void TEMPL<T1,T2>::initAllocObj()                     \
  { sAllocObj = new APT_FixedSizeAllocator(sizeof(TEMPL<T1,T2>)); }\
  APT_FixedSizeAllocator* TEMPL<T1,T2>::sAllocObj


inline void* APT_FixedSizeAllocator::alloc()
{
#if DEBUG_FAST_ALLOC > 1
  validate();
#endif

  Link* answer = freeList_;
  if (!answer)
  { alloc_();
    answer = freeList_;
  }
  freeList_ = freeList_->next_;

#if defined(DEBUG_FAST_ALLOC)
  assert(((unsigned long)answer&7) == 0);
  assert(eltsFree_ > 0);
  -- eltsFree_;
  ++ eltsOutstanding_;
#endif

  return answer;
}
```

```
inline void APT_FixedSizeAllocator::free(void* elt)
{

#if DEBUG_FAST_ALLOC > 1
  validate();
#endif

  if (!elt) return;

// Squash freed mem to flush out accesses to it
#ifndef APT_NODEBUG
  memset(elt, (int)'?', eltSize_);
#endif

#if defined(DEBUG_FAST_ALLOC)
  // freeList_ empty iff eltsFree_ empty
  assert((eltsFree_ > 0) ^ (freeList_ == 0));
  ++ eltsFree_;
  assert(eltsOutstanding_ > 0);
  -- eltsOutstanding_;
#endif

  if (sDisable) {
    ::operator delete(elt);
    return;
  }

  Link* lnk = (Link*) elt;
  lnk->next_ = freeList_;
  freeList_ = lnk;
}

#ifdef DEBUG_FAST_ALLOC
// Nonlinear (expensive) consistency check
inline void APT_FixedSizeAllocator::validate()
{
  unsigned long elts;
  Link* lptr;
  for (elts = 0, lptr = freeList_;
       lptr;
       ++elts, lptr = lptr->next_) {
    for (unsigned long ei = sizeof(Link);
         ei < eltSize_;
         ++ei)
      assert(((char*)lptr)[ei] == '?');
  }
  assert(elts == eltsFree_);
}
```

```
#endif


class APT_VariableSizeAllocator
{
public:
  APT_VariableSizeAllocator(APT_UInt32 limit=1024);
  /*
     effect    Constructs an allocator that will allocate elements
               of variable size.
               The limit arg determines a size cutoff above which the
               global heap allocator is used.
  */

  void* alloc(APT_UInt32 sz);
  /*
     effect    Returns storage for sz bytes.  The storage will
               be aligned as 0 (mod 8).
     note      As needed, the system allocator is called to obtain a
               chunk of storage that is divided into several
               sz-sized parcels.
  */
  void free(void* elt, APT_UInt32 sz);
  /*
     effect    Places the indicated elt onto this allocator's
               freelist.
               If elt is 0 there is no effect.
     note      free() does not necessarily return storage to the
               global heap.
     requires  elt must have been returned from this allocator's
               alloc(sz), and not yet free()d.
  */

  void* alloc_memo(APT_UInt32 sz);
  /*
     effect    Like alloc(), except the size of the allocation
               is memoized so that the overload of free() without
               the APT_UInt32 arg may be used.
  */
  APT_UInt32 free(void* elt);
  /*
     effect    Like free(void*, APT_UInt32), except the allocation size
               is retrieved from the allocation-time memoization.
               The retrieved allocation size is returned.
     requires  elt must have been returned from this allocator's
               alloc_memo(sz), and not yet free()d.
  */
```

```
private:
  APT_VariableSizeAllocator(const APT_VariableSizeAllocator&);
  APT_VariableSizeAllocator& operator= (const APT_VariableSizeAllocator&);

  void alloc_(int bin);

  // size 0-8: bin 0; size 9-16: bin 1; etc.
  int binFromSize(APT_UInt32 sz) const
  { if (sz == 0) return 0;
    return (sz-1)>>3;
  }

  // size is (bin+1)*8
  APT_UInt32 sizeFromBin(int bin) const
  { return (bin+1)<<3;
  }

  struct Link
  {
    Link* next_;
  };

  APT_UInt32 limit_;
  Link* *freeList_;

#if defined(DEBUG_FAST_ALLOC)
  void validate() const;
  unsigned long *eltsFree_;             // freeList_ length
  unsigned long *eltsOutstanding_;      // alloc() - free() count

  // Static member that validates all allocators
  // for use by debuggers
  static void validate_all();
  APT_VariableSizeAllocator* next_;
#endif
};

inline void* APT_VariableSizeAllocator::alloc(APT_UInt32 sz)
{
#if DEBUG_FAST_ALLOC > 1
  validate();
#endif

  if (sz >= limit_)
    return ::operator new(sz);
  int bin = binFromSize(sz);
  Link* answer = freeList_[bin];
  if (!answer)
```

```
  { alloc_(bin);
    answer = freeList_[bin];
  }
  freeList_[bin] = freeList_[bin]->next_;

#if defined(DEBUG_FAST_ALLOC)
  assert(((unsigned long)answer&7) == 0);
  assert(eltsFree_[bin] > 0);
  -- eltsFree_[bin];
  ++ eltsOutstanding_[bin];
#endif

#if DEBUG_FAST_ALLOC > 1
  validate();
#endif

  return answer;
}

inline void APT_VariableSizeAllocator::free(void* elt, APT_UInt32 sz)
{

#if DEBUG_FAST_ALLOC > 1
  validate();
#endif

  if (!elt) return;

// Squash freed mem to flush out accesses to it
#ifndef APT_NODEBUG
  memset(elt, (int)'?', sz);
#endif

  if (sz >= limit_)
    ::operator delete(elt);
  else
  {
    int bin = binFromSize(sz);

#if defined(DEBUG_FAST_ALLOC)
    // freeList_ empty iff eltsFree_ empty
    assert((eltsFree_[bin] > 0) ^ (freeList_[bin] == 0));
    ++ eltsFree_[bin];
    assert(eltsOutstanding_[bin] > 0);
    -- eltsOutstanding_[bin];
#endif

    Link* lnk = (Link*) elt;
```

```
    lnk->next_ = freeList_[bin];
    freeList_[bin] = lnk;
  }

#if DEBUG_FAST_ALLOC > 1
  validate();
#endif
}

inline void* APT_VariableSizeAllocator::alloc_memo(APT_UInt32 sz)
{
  APT_UInt32 allocSz = sz+8;
  void* mem = alloc(allocSz);
  *((APT_UInt32*) mem) = allocSz;
  return ((char*) mem) + 8;
}
inline APT_UInt32 APT_VariableSizeAllocator::free(void* elt)
{
  if (!elt) return 0;
  char* mem = ((char*) elt) - 8;
  APT_UInt32 allocSz = *((APT_UInt32*) mem);
  free(mem, allocSz);
  return allocSz-8;             // return original alloc_memo's sz
}

#endif                          // APT_FAST_ALLOC_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_FILESET_H
#define APT_FILESET_H

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_HOSTFILENAME_H
#include <apt_util/hostfilename.h>
#endif

#ifndef APT_SCHEMA_H
#include <apt_framework/schema.h>
#endif

class APT_ParseError;
class APT_Node;
class APT_NodeSet;
class APT_ErrorLog;
class APT_String;
class istream;
class ostream;


class APT_FileSet
{
public:
  APT_FileSet();
  /* makes an empty file set */

  ~APT_FileSet();

  APT_FileSet(const APT_FileSet&);
  APT_FileSet& operator= (const APT_FileSet&);
  // supports value semantics

  bool hasSchema() const;
  /*
     effect    Tells if this file set has an external schema.
  */
```

```
APT_Schema schema() const;
/*
   effect    Returns this file set's external schema.
   requires  hasSchema() must be true.
*/

void setSchema(const APT_Schema&);
void unSetSchema();
/*
   effect    Sets (or removes) this file set's external schema.
*/

int numLogicalFiles() const;
/*
   effect    Tells how many logical files are listed in this file set.
   note      A logical file consists of one or more physical files,
             all on the same node.  The physical files, taken in
             sequence, represent the content of the logical file.
             The break between physical files occurs at a record
             boundary.
*/

APT_String nodeContainingLogicalFile(int logicalFile) const;
/*
   effect    Returns the name of the node that contains the given
             logical file.
   note      Logical files are ordered in order of appearance in the
             file set.
   requires  0 <= logicalFile < numLogicalFiles()
*/

int numPhysicalFiles(int logicalFile) const;
/*
   effect    Tells how many phsical files comprise the given logical
             file.
   requires  0 <= logicalFile < numLogicalFiles()
*/

APT_HostFileName entry(int logicalFile, int physicalFile) const;
/*
   effect    Retrieves the indicated node/filename entry.
   note      Physical files are ordered in order of appearance
             within a logical file entry.
   requires  0 <= logicalFile < numLogicalFiles()
             0 <= physicalFile < numPhysicalFiles(logicalFile)
*/
```

```
bool hasSameNodes(const APT_FileSet&) const;
/*
   effect    Tells if the two file sets have the same vector of
             nodes as returned by nodeContainingLogicalFile().
*/

APT_Node* makeNodeMap(int* numberOfPartitions, APT_ErrorLog& log) const;
/*
   effect    Builds a node map suitable for use in
             APT_Operator::setNodeMap().
             The numberOfPartitions is set to the length of the
             returned vector (this is the number of logical files that
             have one or more physical files).
             The client must delete[] the returned vector.
             Returns null (and sets *numberOfPartitions to zero) if
             there are no physical files.
   note      The node map will have one entry per "live" logical
             file, even if multiple logical files are on the same
             node.
             The node map is ordered in the order of appearance of
             logical files in the file set.
   errors    If any of the node names cannot be found in the config
             file, an error is reported in the log arg, and null is
             returned.
   requires  numberOfPartitions non-null.
*/

void setNumLogicalFiles(int num);
/*
   effect    Sets the number of logical files to be accommodated by
             this file set.
   note      If the number of logical files is increased, existing
             logical file entries are preserved, and the new logical
             files will each initially have 0 physical files.
             If the number of logical files is decreased, then
             high-numbered logical file entries are accordingly
             discarded.
   requires  num >= 0
*/

void addPhysicalFile(int logicalFile, const APT_HostFileName& hfn,
                     APT_String* errp=0);
/*
   effect    Adds the indicated node/filename entry to this file
             set, associating it with the indicated logical file.
   requires  0 <= logicalFile < numLogicalFiles()
             The given hfn must not already be part of the specified
             logical file; this is reported in errp if provided.
```

```
            The supplied hfn must be valid.
            The hfn must have a host name component.
            The host name component must not be "*".
            All physical files added to the same logicalFile must
            have the same host name; this is reported in errp if
            provided.
*/

void createFileList(const char* namePattern,
                    int numLogicalFiles, int physicalFilesPerLogicalFile,
                    const APT_Node* nodeMap,
                    const APT_String& diskPool,
                    APT_ErrorLog& log);
/*
   effect   Generates a list of files for this file set.
            For each node in the nodeMap, a logical file is
            created, with the indicated number of physical files
            per logical file.
            For each logical file, the corresponding node's
            resource definitions are consulted to find disk
            definitions in the indicated diskPool.  These disks are
            used (round-robin) to allocate the physical files of a
            logical file.
            If the physicalFilesPerLogicalFile parameter is zero,
            then the number of matching disk resources for each
            node determines the number of physical files for that
            node's logical file.
            The namePattern is used to generate the physical file
            names of the form:

              <node>:/dir/NAME

            Where <node> is the nodename from the nodeMap, and
            /dir/ is the matching disk entry for that node used to
            store this physical file.  The NAME portion is
            constructed from text supplied in namePattern, where
            the following substitutions are available:

              %u: replaced by username
              %d: replaced by current date in YYYYMMDD format
              %t: replaced by current time in HHMMSS format
              %p: replaced by partition number (logical file number)
              %n: replaced by physical file number within logical file
              %h: replaced by a 7-character hash (see below)
              %%: replaced by single % character

            The %h component is available if the client cannot
            assure that no runtime file name collision will occur.
```

```
                If present, %h will be replaced by 7 characters
                representing a 32-bit hash value contructed from:
                process ID, hostname, microsecond time, entry number
                within file set, and an in-process file set ticket.
                The intention is that the %h hash, alone or in
                conjunction with other substitutions such as %d date
                and %t time, will help reduce the probability of a file
                name collision.
                The hash characters are chosen from the set a-z,0-5.
   note         Any previous list of files associated with this file
                set object is replaced.
   errors       Any format problems in namePattern are noted in the log
                object.
                If a node (logical file) does not have any disk
                resources in the given diskPool, then that logical file
                will have no physical files allocated to it.  A warning
                is noted in the log object in this case.
   requires     namePattern non-null
                namePattern must contain only valid %-substitutions
                numLogicalFiles >= 1
                physicalFilesPerLogicalFile >= 0
                nodeMap non-null
                The supplied nodeMap must have exactly numLogicalFiles
                entries.
*/

void setTestMode(bool test) { testMode_ = test; }
/*
   effect       If set to true, causes the various substitutions in
                createFileList() to use static values.
*/


// I/O

void parse(const char*, APT_ParseError* err=0);
void parse(istream&, APT_ParseError* err=0);
/*
   effect       Parses the input, expecting a file set.
                If the parse is successful, this APT_FileSet
                object's state is replaced by the parsed file set
                description.
   syntax       --Orchestrate File Set v1
                --LFile
                node:/filename
                node:/another_filename
                ...
                --LFile
```

```
                ...
                --Schema                // optional
                record (...)
    old syntax
                node:/filename
                ...
                --APT_Schema            // optional
                record (...)

                Until the schema section, lines are limited to 1k in
                length.
    throws      APT_ParseError: trouble parsing the supplied input as a
                  file set.  If the err argument is 0, then
                  the error is thrown; if err is non-null, then the
                  error object is copied into the object pointed to by
                  err.
    requires    string arg non-null
  */

  void unparse(ostream&) const;
  APT_String unparse() const;
  /*
     effect     Prints or returns a parse()able representation of
                this APT_FileSet.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_FileSet&);

private:
  bool hasSchema_;
  APT_Schema schema_;

public:
  struct LogicalFile
  {
    LogicalFile();
    ~LogicalFile();
    LogicalFile(const LogicalFile&);
    LogicalFile& operator= (const LogicalFile&);

    friend APT_Archive& operator|| (APT_Archive&, LogicalFile&);

    APT_String nodeName_;
    int numPhysical_;
    APT_HostFileName* physical_;
  };
private:
```

```
  int numLogical_;
  LogicalFile* logical_;
  bool testMode_;
};
APT_DIRECTIONAL_SERIALIZATION(APT_FileSet);

#endif // APT_FILESET_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_IDENTIFIER_H
#define APT_IDENTIFIER_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#include <string.h>
#include <apt_util/condition.h> // instead of <strings.h>
#include <ctype.h>

class APT_Archive;
class istream;
class ostream;

class APT_Identifier
/* Class for representing Torrent-style identifiers.

   An Torrent-style identifier is a string that:

   - contains no whitespace
   - begins with a letter or underscore
   - contains only letters, underscores, and digits
   - is compared case-insensitively (but case is preserved)
*/
{
public:
  APT_Identifier() : str_(&sNullChar), len_(0) {}
  /*
     effect    Makes empty identifier, a degenerate (but legal) case.
     note      Most abstractions that take an identifier will complain
               if given an empty identifier.
  */
  APT_Identifier(const APT_Identifier& rhs);
  APT_Identifier& operator= (const APT_Identifier& rhs);
  ~APT_Identifier() { if (len_) dtor(); }

  APT_Identifier(const char*); // convert from null-terminated string
  APT_Identifier(const char*, APT_UInt32 len); // convert from buffer and length
  APT_Identifier(const APT_String&);
```

```
  /*
     effect    Constructs this identifier, checking its content for
               legality.
  */

  APT_Identifier& operator= (const char*);
  APT_Identifier& operator= (const APT_String&);
  // checks content for legality

  APT_String string() const { return str_; }
  /*
     effect    Makes a copy of this identifier as a string.  Note that
               data() is more efficient.
  */

  APT_UInt32 length() const { return len_; }

  /* access underlying string buffer (caution: this buffer goes away
     when this identifier is destroyed or assigned to). */
  const char* data() const { return str_; }
  operator const char* () const { return str_; }

  // all the == and != ops are case-insensitive
  friend inline bool operator== (const APT_Identifier&, const APT_Identifier&);
  friend bool operator!= (const APT_Identifier& l, const APT_Identifier& r)
    { return !(l == r); }

  // add some overloads to avoid certain ambiguities
  friend inline bool operator== (const APT_Identifier& lhs, const char* rhs);
  friend bool operator== (const char* lhs, const APT_Identifier& rhs)
  { return rhs == lhs; }

  friend bool operator!= (const APT_Identifier& l, const char* r)
    { return !(l == r); }
  friend bool operator!= (const char* l, const APT_Identifier& r)
    { return !(l == r); }

  friend APT_Archive& operator|| (APT_Archive&, APT_Identifier&);

  friend ostream& operator<< (ostream&, const APT_Identifier&);
  friend istream& operator>> (istream&, APT_Identifier&);

  static bool isValid(const char*); // null-terminated
  static bool isValid(const char*, APT_UInt32);
  // tells if the given string is a legal identifier

  void assignFrom_nocheck(const char*, APT_UInt32);

private:
  static char* dup(const char*, APT_UInt32 len);
  void dtor();
```

```
  static char sNullChar;

  char* str_;                   // null-terminated string rep
  APT_UInt32 len_;
};
inline APT_Archive& operator<< (APT_Archive& ar, const APT_Identifier& d)
  { return ar || (APT_Identifier&) d; }
inline APT_Archive& operator>> (APT_Archive& ar, APT_Identifier& d)
  { return ar || d; }

inline bool operator== (const APT_Identifier& i1, const APT_Identifier& i2)
{ if (i1.len_ != i2.len_) return false;  // get lucky?
  return (strcasecmp(i1.str_, i2.str_) == 0) ? true : false;
}
inline bool operator== (const APT_Identifier& i1, const char* i2)
{ if (tolower(i1.str_[0]) != tolower(i2[0])) return false; // get lucky?
  return (strcasecmp(i1.str_, i2) == 0) ? true : false;
}

#endif // APT_IDENTIFIER_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1998 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_KEYGROUP_H
#define APT_KEYGROUP_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

class APT_FieldSelector;
class APT_PropertyList;
class APT_InputAccessorInterface;
class APT_OutputAccessorInterface;
class APT_InputInterface;
class APT_OutputInterface;
class APT_FieldTypeDescriptor;
class APT_OutputAccessorBase;
class APT_Schema;
class APT_ErrorLog;
class ostream;

class APT_KeyGroupRep;

class APT_KeyGroup : public APT_Persistent
/* Abstraction for treating one or more input fields as keys for the
   purposes of hashing, equality testing, or ordered comparisons.

   This abstraction is used as follows:

   1. A view-adapted schema is supplied.  This is generally the
      view-adapted input schema of the operator that is using the
      keygroup object.

   2. One or more key fields is defined.  These keys must be present
      in the view-adapted schema (but note that the view-adapted
      schema can be supplied after the key fields have been specified;
      in any case, validate() checks that key fields are present in
      the view adapted schema).

   3. A mode of operation is specified, either equality/hash or
      ordered comparison.  This controls which generic functions will
      be instantiated for each key, and governs certain error checks.

   4. The key group object is validated.  All error checking occurs
      here, and any relevant warnings are generated.
```

5. An interface schema is extracted. This will be a subset of the view-adapted schema, corresponding to the key fields being used. The operator using this keygroup object generally uses this as the input interface for the input data set this keygroup will be working with.

6. The keygroup object is setup with an input cursor (or input interface object). The keygroup object uses this opportunity to set up input accessors for its key fields.

7. One or more Value objects are bound to this keygroup; such Value objects reflect this keygroup's current key values (as established by the underlying input cursor or interface).

   Value objects may be copied and assigned, in which case the destination Value becomes a static snapshot of the source Value object's key values.

8. The operator advances its input cursor, and uses operations on Value object(s) to perform hashing/equality operations (if the mode so permits), or ordered comparison operations (again, if the mode was so set).

9. In some cases an operator may wish to write out keygroup values. This is done by defining a field name to be used for output when addKey is called. validateOutput() must be called with the output interface schema, and setup() must be called with the output cursor or interface. Then write() may be called on keygroup values associated with the keygroup.

```
*/
{
public:
  APT_KeyGroup();
  ~APT_KeyGroup();

  void setViewAdaptedSchema(const APT_Schema&);
  APT_Schema viewAdaptedSchema() const;
  /*
     effect    Accesses the view adapted schema against which this
               keygroup's keys will be bound.
               Initially empty.
  */

  void addKey(const APT_FieldSelector&, const APT_PropertyList& param);
  /*
     effect    Specifies the name of a key field, together with any
               parameters to be passed to its compare/equality
               function.
     note      For compare(), the order in which keys are added is
               significant.
```

```
              The function=funcname property is recognized, and is
              used to select the compare/equality function.
              The order=ascending/descending property is recognized,
              and is used to reverse the sense of ordered
              comparison (order=ascending is the default).
              The nulls=first/last property is recognized,
              and is used to sort nulls before or after values
              (nulls=first is the default).
   requires   The key must not be a vector field.  TBD: support
              vectors?
              The key must be a top-level field of the
              viewAdaptedSchema().
              A given key field may not be added multiple times.
*/

void addKey(const APT_FieldSelector& inKey,
            const APT_FieldSelector& outKey,
            const APT_PropertyList& param);
/*
   effect     Specifies the name of a key field, together with any
              parameters to be passed to its compare/equality
              function, as well as the name of a field that will be
              used to write the value of the field when the Value
              type's write() function is called.
   note       It is not required that any or all fields in a key group
              have output key fields.
              An outKey's type (in the schema passed to
              validateOutput()) must be the same as the field's type
              in the input viewAdaptedSchema().
   requires   See addKey() above.
              outKey must be a field (not necessarily top-level) in
              the schema passed to validateOutput().
*/
void addKey(const APT_FieldSelector& inKey,
            const APT_PropertyList& param,
            const APT_FieldTypeDescriptor* td);
void addKey(const APT_FieldSelector& inKey,
            const APT_FieldSelector& outKey,
            const APT_PropertyList& param,
            const APT_FieldTypeDescriptor* td);
// deprecated forms to support obsolete user-specified type
// Revert to common forms above if td is 0

int numKeys() const;
void getKey(int index, APT_FieldSelector*, APT_PropertyList* param) const;
void getKey(int index, APT_FieldSelector* inKey, APT_FieldSelector* outKey,
            APT_PropertyList* param) const;
/*
   effect     Accesses the key(s) that have been defined on this
              keygroup.  The field name and param list are written
              into the arguments, if non-null.
```

```
   requires  0 <= index < numKeys()
             For the second overload form, one of the output-key
             forms of addKey() must have been used.
*/

const APT_FieldTypeDescriptor* getKeyType(int index) const;
// for testing deprecated addKey() forms

enum Mode
{
  eComparison,                // compare() usable
  eEquality                   // hash() and isEqual()
};

void setMode(Mode);
Mode mode() const;
/*
   effect    Accesses this keygroup's mode of operation.  Default is
             eComparison.
*/

/* the above information (viewAdaptedSchema, keys, mode) is
   serialized by the class' persistence implementation.  The
   validate()d and setup() state is not serialized. */

void copyInitialization(APT_KeyGroup* dest) const;
/*
   effect    Modifies dest to have the same viewAdaptedSchema, keys,
             and mode as this keygroup.
*/

void validate(APT_ErrorLog&);
/*
   effect    The generic functions corresponding to the key fields
             are instantiated and parameterized.
             Any errors or warnings are accumulated into the err
             argument.
   requires  setViewAdaptedSchema() must have been called.
*/

void validateOutput(const APT_Schema & outputSchema,
                    APT_ErrorLog&);
/*
   effect    Verify that outputSchema contains fields of the
             names and types required.
             Any errors or warnings are accumulated into the err
             argument.
   requires  setViewAdaptedSchema() must have been called.
             validate() must have been error-free.
*/
```

```
APT_Schema interfaceSchema() const;
/*
   effect    Returns an input interface schema which is the subset
             of the viewAdaptedSchema() corresponding to our keys.
   requies   validate() must have been error-free.
*/

void setup(APT_InputAccessorInterface* cur);
/*
   effect    Uses the supplied cursor to setup the input accessors
             that will be used by this keygroup.
   requires  cur non-null
             cur->isSetup() is true
             validate() must have been error-free
             The cursor must be bound to an interface that corresponds
             the viewAdaptedSchema().
*/
void setup(APT_InputInterface* intf);
/*
   effect    Uses the supplied interface to setup the input accessors
             that will be used by this keygroup.
   requires  intf non-null
             intf->isBound() is true
             validate() must have been error-free
             The interface must correspond to the viewAdaptedSchema().
*/

void setup(APT_OutputAccessorInterface* cur);
/*
   effect    Uses the supplied cursor to setup the output accessors
             that will be used to write values by this keygroup.
   requires  cur non-null
             cur->isSetup() is true
             validate() must have been error-free
             validateOutput() must have been error-free
             The cursor must be bound to an interface that corresponds
             the schema passed to validateOutput().
*/
void setup(APT_OutputInterface* intf);
/*
   effect    Uses the supplied interface to setup the output accessors
             that will be used to write values by this keygroup.
   requires  intf non-null
             intf->isBound() is true
             validate() must have been error-free
             validateOutput() must have been error-free
             The interface must be bound to an interface that corresponds
             the schema passed to validateOutput().
*/

void enablePrinting(APT_ErrorLog&);
```

```
/*
   effect    Enables Value::print() and/or Value::scan() to be used.
   warnings  Issued if one or more key fields do not have a print
             function.
   requires  validate() must have been error-free.
*/

class Value
{
public:
  Value();
  // not usable until assigned to or trackCurrentValue() called

  Value(const APT_KeyGroup* g);
  // equivalent to default ctor followed by trackCurrentValue(g)

  Value(const APT_KeyGroup* g, bool tracking);
  /* associates this Value with the given keygroup object; if
     tracking is true then this Value is in tracking mode; if
     tracking is false then this value is a non-tracking value */

  ~Value();
  /* requires  The keygroup with which this Value is associated
               must still exist.
  */

  Value(const Value&);
  Value& operator= (const Value&);
  /*
     effect    Copies the value.  If the argument is tracking a
               keygroup's current value, then that current value is
               copied into this Value.  After the copy/assign, this
               Value does *not* track the keygroup's current value.
     requires  If the argument Value is tracking a keygroup's
               current value, then that keygroup object must have
               been validate()d and setup(), and its cursor or
               interface must have a current record.
  */

  friend APT_Archive &operator||(APT_Archive &ar, APT_KeyGroup::Value& v);
  /*
    effect    serializer for values.  If the Value is tracking, only
              the tracking relationship is serialized.  If it's been
              copied or assigned to (and therefore is not tracking),
              then the value itself is serialized.
    Note      When loading, the Value must already have been bound
                    to a KeyGroup, e.g. using trackCurrentValue() or being
              assigned to.
  */

  void trackCurrentValue(const APT_KeyGroup*);
```

```
   /*
      effect    Modifies this Value to track the supplied keygroup's
                current value.
      note      This Value's implementation refers back to the
                argument APT_KeyGroup object.  Therefore, the
                argument APT_KeyGroup should not be destroyed before
                this Value object.
                If this Value object is copied or assigned to another
                Value, the destination gets a copy of this Value's
                current value.  The copy/assign destination does not
                track the keygroup's current value.
      requires  The supplied keygroup object has been validate()d.
   */

   bool hasValue() const;
   /*
     returns   True if this Value has been bound with an assignment
               or trackCurrentValue(), that is, all the useful
               functions are valid.  False otherwise.
   */

   static APT_UInt32 hash(const Value&);
   static unsigned hash_rw(const Value&); // signature appropriate for RW hash
   /*
      effect    Computes a hash of the indicated Value.
      note      This function is static to facilitate use of hash
                tables that take an ordinary (non-member) function
                pointer.
      requires  The Value object must either have had a value
                copied in, or must be tracking a keygroup's current
                value.  In the latter case, the keygroup must have a
                current value.
                The keygroup's mode() must be eEquality.
   */

   static bool isEqual(const Value& left, const Value& right);
   /*
      effect    Determines if the two Values are equal.
      requires  Both Value objects must either have had a value
                copied in, or must be tracking a keygroup's current
                value.  In the latter case, the keygroup must have a
                current value.
                The Value objects must have the same configuration
                (tied to keygroups with same schema, key
                definitions).
      note      Two null keys are considered to be equal.
                       this uses the types' eqFunc if in eEquality mode,
                otherwise uses compare() == eEqual.
   */
   friend bool operator== (const Value& left, const Value& right)
   { return isEqual(left, right); }
```

```
   friend bool operator!= (const Value& left, const Value& right)
   { return !isEqual(left, right); }
   friend bool operator< (const Value& left, const Value& right)
   { return (compare(left, right) == APT_KeyGroup::Value::eLessThan); }
   friend bool operator> (const Value& left, const Value& right)
   { return (compare(left, right) == APT_KeyGroup::Value::eGreaterThan); }
   friend bool operator<= (const Value& left, const Value& right)
     { return ! (left > right); }
   friend bool operator>= (const Value& left, const Value& right)
     { return ! (left < right); }

   enum CompareResult { eLessThan=-1, eEqual=0, eGreaterThan=1 };
   static CompareResult compare(const Value& left, const Value& right);
   /*
      effect    Compares the two keygroups.
      requires  Both Value objects must either have had a value
                copied in, or must be tracking a keygroup's current
                value.  In the latter case, the keygroup must have a
                current value.
                Both keygroup's mode() must be eComparison.
                The Value objects must have the same configuration
                (tied to keygroups with same schema, key
                definitions).
   */

   void print(ostream& ostr, bool namePrefix=true) const;
   /*
      effect    Prints this Value.  If namePrefix is true, then the
                following format is used:

                  name:value name:value ...

                If namePrefix is false, then the field name and colon
                are not printed.  A trailing space is not printed.
      note      print()ing is not necessarily fast.
      requires  This Value's keygroup's enablePrinting() function
                must have been called.
   */

   APT_Status scan(const APT_String* literalVec);
   /*
      effect    Scans the supplied vector of literal strings (one for
                each component field of this Value) to initialize this
                Value.
                An empty string component of literalVec indicates
                that the corresponding field should be null.
      returns   Whether the scan was successful or not.  If unsuccessful,
                the value(s) written to this Value is unspecified.
      requires  This Value's keygroup's enablePrinting() function
                must have been called.
                This Value must not be in tracking mode (it must be a
```

```
                free-standing copy).
    */

    void write() const;
    void write_bind() const;
    /*
       effect    Writes this Value using the fields defined in the
                 keygroup.
       note      In the second form,
                 APT_FieldTypeDescriptor::bindValueType() is used, not
                 copyValueType(), so the Value must not change between the
                 time write() is called and putRecord() is called for the
                 cursor associated with the accessor(s).
       requires  This Value must have been set (either by trackCurrentValue
                 or assignment).
    */
  private:
    void dtor();
    void copy(const Value& rhs);

    void updateTracked() const
    { if (tracking_) updateTracked_(); }

    void updateTracked_() const;

    const APT_KeyGroupRep* keyGroup_;
    bool tracking_;

    APT_UInt32 hashVal_;        // non-zero is a memoized value

    struct KeyValue
    {
      KeyValue() : value_(0), isNull_(false) {}

      void* value_;
      bool isNull_;

      APT_DECLARE_NEW_AND_DELETE(KeyValue);
    };
    KeyValue* keyValues_;

    APT_DECLARE_NEW_AND_DELETE(Value);
  };

private:
  friend class Value;

  // prohibit copy/assign
  APT_KeyGroup(const APT_KeyGroup&);
  APT_KeyGroup& operator= (const APT_KeyGroup&);
```

```
  APT_KeyGroupRep* rep_;

  APT_DECLARE_RTTI(APT_KeyGroup);
  APT_DECLARE_PERSISTENT(APT_KeyGroup);
};
APT_DIRECTIONAL_SERIALIZATION(APT_KeyGroup::Value);

#endif // APT_KEYGROUP_H
```

```
// -*-Mode: C++-*-
//#
//#  Copyright (C) 1996 Torrent Systems, Inc.
//#  All Rights Reserved
//#
//#  $Id: locator.h,v 1.2 1998/05/15 16:50:54 smr Exp $

// Representation of a generalized locator.  The syntax is
// [[protocol]:]location.  The protocol and colon
// delimiter are optional; if no protocol is specified it defaults to
// file access.  If the location information contains a colon, the
// leading colon must be specified.

#ifndef APT_LOCATOR_H
#define APT_LOCATOR_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

class APT_String;
class APT_Identifier;
class APT_LocatorImpl;

class APT_Locator : public APT_Persistent {
  APT_DECLARE_PERSISTENT(APT_Locator);
  APT_DECLARE_RTTI(APT_Locator);

public:

  APT_Locator(const char *info);
  /*
   * Constructs a new APT_Locator from the specified string.
   */

  APT_Locator (const char *protocol, const char *location);
  /*
   * Constructs a new APT_Locator with the given protocol and location.
   */

  APT_Locator();
  /*
   * A default constructed APT_Locator is invalid.
   */
```

```
  APT_Locator(const APT_Locator &);
  APT_Locator &operator=(const APT_Locator &);
  ~APT_Locator();

  bool operator==(const APT_Locator &) const;
  bool operator!=(const APT_Locator & rhs) const
  { return !(*this == rhs); }

  bool isValid() const;
  /*
   * Is this Locator valid.  To be valid, the protocol must be a legal
   * token.
   */

  APT_String errorCondition() const;
  /*
   * Error that occurred during construction.
   */

  APT_String location() const;
  /*
   * Returns the location portion of the locator.
   * isValid() must be true.  If isValid() is false, a requirements violation
   * will be signalled that will contain the same information that
   * errorCondition() would contain.
   */

  APT_Identifier protocol() const;
  /*
   * Returns the protocol portion of the locator.  If none was specified
   * in construction, the locator will be "file".
   * isValid() must be true.  If isValid() is false,
   * a requirements violation will be signalled that will contain the
   * same information that errorCondition() would contain.
   */

  APT_String unparse() const;

private:
  APT_LocatorImpl *rep_;

  friend ostream& operator<< (ostream&, const APT_Locator&);
  friend istream& operator>> (istream&, APT_Locator&);

};

#endif
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_LOGMSG_H
#define APT_LOGMSG_H

#include <strstream.h>

#ifndef APT_ERRIND_H
#include <apt_util/errind.h>
#endif

#ifndef APT_ERRORCONFIG_H
#include <apt_util/errorconfig.h>
#endif

/**** message logging utilities ****/

/* OBSOLETE: Use APT_ErrorLog (in errlog.h) for this functionality.
 * If you don't have a log object handy, thread one in.
 */

#define APT_DETAIL_LOGMSG_VERYLONG(severity_, module_, errorIndex_, message_, ident_)
\
  do {                                                                  \
      ostrstream _stream_;                                              \
      _stream_ << message_ << ends;                                     \
      APT_ErrorConfiguration::get().output(module_, severity_, errorIndex_, \
                                  _stream_.str(), ident_);              \
      delete [] _stream_.str();                                         \
  } while (0)

#define APT_DETAIL_LOGMSG_LONG(severity_, module_, errorIndex_, message_) \
  do {                                                                  \
      ostrstream _stream_;                                              \
      _stream_ << message_ << ends;                                     \
      APT_ErrorConfiguration::get().output(module_, severity_, errorIndex_, \
                                  _stream_.str(), "");                  \
      delete [] _stream_.str();                                         \
  } while (0)

#define APT_DETAIL_LOGMSG(severity_, errorIndex_, message_)             \
  APT_DETAIL_LOGMSG_LONG(severity_, APT_localErrorSourceModule,         \
                         errorIndex_, message_)

// Leave old time log function around for backwards compatibility of some
// code.
extern APT_String APT_LogTime();

#endif // APT_LOGMSG_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_PERSIST_H
#define APT_PERSIST_H

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_RTTI_H
#include <apt_util/rtti.h>
#endif

#include <new.h>

class APT_SerializedBases;

class APT_Persistent
/* Abstract base class for "complex persistent" classes.  See
   persistence.html for an overview of the persistence system.

   Note that it is not necessary to derive a class from
   APT_Persistent in order for it to be persistent.  A class
   can be made "simple persistent" just by defining the serialization
   overloads: operator|| (APT_Archive&, T&) etc.

   However, there are circumstances under which simple persistence is
   inadequate.  A class should be complex-persistent if any of the
   following conditions are met:

   - Clients want to serialize pointers to this class in contexts
     where multiple such pointers to the same object might be
     serialized, or
   - The class contains complex-persistent sub-structure with the
     possibility of structural sharing, or
   - Polymorphic persistence is needed, or
   - Data versioning support is desired.

   For a class to be complex-persistent, the following rules must all
   be followed:

   1.  A class must publicly inherit from APT_Persistent,
       direcly or indirectly, exactly once.  APT_Persistent may
       be a virtual base class.
   2.  Concrete classes must use the APT_DECLARE_PERSISTENT
```

```
       and APT_IMPLEMENT_PERSISTENT macros.
       Abstract classes must use the APT_DECLARE_ABSTRACT_PERSISTENT
       and APT_IMPLEMENT_ABSTRACT_PERSISTENT macros.
   3.  Provide RTTI for your class using appropriate APT_DECLARE_RTTI
       and APT_IMPLEMENT_RTTI_xxx macros.  Each complex-persistent base
       class must be listed as an RTTI base of your class.
   4.  A default constructor must exist, unless the ABSTRACT form of
       the macros is used.  The default constructor need not be
       public; it may be protected or private if appropriate.  A
       compiler-generated default constructor is acceptable.
   5.  A serialize(APT_Archive&, APT_UInt8) function must be
       implemented in all derivations, regardless of whether the
       derived class is abstract or concrete.  Note that the
       persistence derivation macros declare this function for you.
       See the description of this function below.
   6.  If the derived class is a template class, then its RTTI class
       name must be defined somewhere.  See rtti.h.

   The persistence facility is not usable at static initialization (STI)
   time.
*/
{
  APT_DECLARE_RTTI(APT_Persistent);

public:
  APT_Persistent();
  virtual ~APT_Persistent();
  APT_Persistent(const APT_Persistent&);
  APT_Persistent& operator= (const APT_Persistent&);

  friend APT_Archive& operator|| (APT_Archive&, APT_Persistent& obj);
  /*
     effect    Serializes the indicated object via its serialize()
                function.
               The entire (derived) referenced object is serialized
               regardless of whether it has been previously serialized
               on the supplied archive.
               When loading, the supplied object is reused as follows:
               - obj's (derived) destructor is called.
               - obj is default-constructed "in place" to its derived
                 type.
               - obj's state is loaded from the archive via its
                 serialize() function.
               As a result, when loading, the serialize() function is
               always called on a default-constructed object.
     throws    APT_Archive::BadData: Check bytes surrounding object's
                 serialization did not load correctly.
```

```
              APT_Archive::Eof: Eof encountered while trying to load
                check bytes.
   requires   When loading, the archive must be positioned to where
              an object of the same (derived) type as obj was stored
              by reference.
*/

friend APT_Archive& operator|| (APT_Archive&, APT_Persistent*& ptr);
friend APT_Archive& operator|| (APT_Archive& ar,
                                const APT_Persistent*& ptr)
  { return ar || (APT_Persistent*&) ptr; }
/*
   effect     Serializes the indicated pointer via the referenced
              object's serialize() function.
              When storing, the entire (derived) object is stored
              along with type information to allow an appropriate
              object type to be created upon loading.
              When loading, either a previously loaded object is
              used, or a new object is dynamically allocated (using
              the stored type information to determine what kind of
              object to create), default-constructed, and loaded via
              its serialize() function.
              The caller is responsible for managing the deletion of
              loaded pointers.
   note       After an object is first stored (either by pointer or
              by reference), the archive remembers that the object
              has been stored in the archive (it memoizes the
              address).  Subsequent stores (via pointer) of the same
              object skip the call to serialize(), and instead store
              a simple marker referencing the previously stored
              object.
              Similarly, when loading, the archive remembers which
              objects have been loaded, so that subsequent pointer
              loads of a memoized object are resolved to the
              previously loaded object.
              Address memoization extends across the serialization of
              a complex-persistent object.  When serialization of a
              top-most complex-persistent object is completed, the
              archive flushes its memoizations.
              When storing, the pointer typically (but not always)
              points to a dynamically allocated object.  When a
              pointer is loaded, the persistence system will bind the
              loaded pointer to a either a previously loaded object,
              or a newly allocated object.
              If a pointed-to object is a member of some other
              object, or is part of an array, then just the
              pointed-to object is stored (unless it has already been
```

```
                stored, in which case its memoization marker is
                stored).  When it is later loaded, it will be
                dynamically allocated as a freestanding object (unless
                it has already been loaded, in which case the pointer
                is made to refer to the loaded sub-object).
                Null pointer values are properly handled.
                If a memoized object to which a loaded pointer should
                refer no longer exists (it has been destroyed), then
                the loaded pointer will be null This is because the
                archive class is not necessarily seekable, so we cannot
                attempt to reload the destroyed object. A warning is
                issued.
    throws      APT_Archive::BadClass:  When loading, the stored
                  object's type name does not match any
                  complex-persistent class name in this program.  The
                  stored object is skipped and the pointer loaded is
                  set to null before the error is thrown.  If the
                  archive's failOnBadClass() flag is false, then no
                  error is thrown, and load processing continues.
                APT_Archive::BadPointer:  This operator is overloaded
                  in each derived persistent class.  When loading, if
                  the loaded object's type does not contain as an
                  accessible base class the class of the pointer being
                  loaded, the pointer is loaded as 0, and this error is
                  thrown.
                APT_Archive::BadData: Check bytes surrounding object's
                  serialization did not load correctly, or other
                  aspects of the loaded data were bad.
                APT_Archive::Eof:  Eof occured during this load
                    operation.
    requires    When loading, the archive must be positioned to where
                a complex-persistent pointer was stored.
                When storing, if ptr refers to an instance of a class
                template, its RTTI classname must have been set
                correctly.
                Must not be called at STI time.
  */

protected:
public:                             /* made public to work around
                                       suspected gcc bug; not intended for
                                       public use! */
  int doThisBase(const APT_TypeInfo&);
  /*
     effect     Called by wrapSerialize() to determine whether it
                should call serialize().  If the answer is "yes", then
                that fact is recorded so a subsequent attempt to
```

```
            serialize the same base (such as could occur with a
            virtual base class) will return "no".
  */

protected:
  APT_Persistent* cloneType() const;
  /*
     effect    Returns a dynamically allocated, default constructed,
               instance of the same class as this object.
               The caller is responsible for deleting the returned
               object.
     note      Since this function is protected, it is up to derived
               classes to expose a public cloning interface
               implemented in terms of this function.  The derived
               class's public clone function can return an object
               pointer of the appropriate derived type.
  */

  APT_Persistent* cloneBySerializing() const;
  /*
     effect    Stores this object into a memory archive, then loads a
               new instance from the memory archive.  The net effect
               is to dynamically allocate an object of the same
               dynamic type as this, loading it via serialization to
               the same state as this object.  A deep copy, albeit an
               expensive one.
               The dynamically allocated copy is returned.
               The caller is responsible for deleting the returned
               object.
     note      Since this function is protected, it is up to derived
               classes to expose a public cloning interface
               implemented in terms of this function.  The derived
               class's public clone function can return an object
               pointer of the appropriate derived type.
  */

  void copyBySerializing(APT_Persistent* dest) const;
  /*
     effect    Similar to cloneBySerialization(), except the dest
               object is loaded (in-place) with this object's stored
               state.
     note      Because the dest argument is loaded in-place, it is
               destroyed, default constructed, and then loaded with
               this object's stored state.
     requires  dest non-null
               The dynamic type of this and dest must be identical.
  */
```

```
  static void loadPointer(APT_Archive&, const APT_TypeInfo& ptrType,
                          void* &rawPtr);
  /*
     effect    Implementation used by derivations' overloads of
               the pointer serialization operators.
  */

  static void storePointer(APT_Archive&, APT_Persistent* ptr);
  /*
     effect    Implementation used by derivations' overloads of
               the pointer serialization operators.
  */

  static int registerClass(const APT_TypeInfo& type,
                           APT_Persistent* (*make)());
  /*
     effect    Registers the supplied make function for the given
               class.
               The return value is always 0.
     requires  class must be unique (not previously registered)
               make non-null
  */

  static void checkPre(APT_Archive&);
  static void checkPost(APT_Archive&);
  /*
     effect    Serializes and integrity-checks a preamble/postamble.
     throws    APT_Archive::BadData: preamble/postamble did not load
                 correctly.
               APT_Archive::Eof: Eof encountered while trying to load
                 preamble/postamble.
  */

private:
  friend class APT_ArchiveRep;  // access instantiate()

  virtual void dispatchSerialize(APT_Archive&)=0;
  /*
     effect    Overridden by the derivation macros at each
               non-abstract level of derivation to call all persistent
               bases' wrapSerialize() function, followed by this
               derivation's wrapSerialize() function.
  */

  virtual void reInit()=0;
  /*
```

```
   effect    Overridden by the derivation macros at each
             non-abstract level of derivation to:
             - explicitly invoke this object's destructor
             - explicitly invoke this object's default constructor
               via placement new.
*/

void* wrapSerialize(void*, int, int&);
/*
   effect    Wraps a call to this derivation level's serialize()
             function.
             The wrapping logic does the following:
             - prevents multiple calls to virtual base class'
               serialize() function.
             - serializes a "data version number".  When loading,
               passes the loaded data version number to the
               serialize() function.  When storing, always stores
               the "current" data version number.
   note      This function's weird signature has to do with how
             dispatchSerialize() uses the RTTI package to cause a
             complex persistent object's entire inheritance diagram
             to be traversed, calling wrapSerialize() at each
             derivation level.
*/

void serialize(APT_Archive&, APT_UInt8 dataVersion);
/*
   effect    Written by the programmer at each level of derivation
             (both abstract and non-abstract).
             This function should serialize this level of
             derivation's state variables.  This class' complex
             persistent base classes will have been serialized
             already.
             If this derivation has any base classes that are not
             complex persistent, it should explicitly serialize
             them, if appropriate.
             When loading, this function always is called on a
             default-constructed object.
             When loading, the dataVersion argument contains the
             data version number that was loaded for this derivation
             level.
*/

void serializeObject(APT_Archive&);
/*
   effect    Wraps a call to dispatchSerialize().
             The wrapping logic sets up protocol used by
```

```
              wrapSerialize() to avoid multiple serializations of
              virtual base classes.
  */

public:                          // for Torrent use only!
  static APT_Persistent* instantiate(const char* type);
  /*
     effect    Returns a dynamically allocated and default-constructed
               object of the given type.
               Returns 0 if the given type does not correspond to any
               complex persistent classes in this program.
     requires  type non-null
  */

private:
  APT_SerializedBases* active_;
  /* data structure used by the implementation to avoid multiple
     serializations of virtual base classes. */
};
APT_DIRECTIONAL_SERIALIZATION(APT_Persistent);

// thankfully, this grungy macro is not for public use
#define APT_DIRECTIONAL_POINTER_SERIALIZATION(T,K)                      \
K APT_Archive& operator<< (APT_Archive& ar, const T* d)                 \
  { ar.checkStoring(); return ar || (T*&) d; }                          \
K APT_Archive& operator>> (APT_Archive& ar, T*& d)                      \
  { ar.checkLoading(); return ar || d; }                                \
K APT_Archive& operator>> (APT_Archive& ar, const T*& d)                \
  { ar.checkLoading(); return ar || (T*&) d; }

APT_DIRECTIONAL_POINTER_SERIALIZATION(APT_Persistent, inline)


/**** example derivation ****/

/*
   // in .h file:
   class Foo : public virtual APT_Persistent
   {
     APT_DECLARE_PERSISTENT(Foo);
     APT_DECLARE_RTTI(Foo);
   public:
     Foo(int, float);
   private:
     Foo();
     int i_;
     float f_;
```

```
   };

   // in .C file:
   APT_IMPLEMENT_PERSISTENT(Foo);
   APT_IMPLEMENT_RTTI_BEGIN(Foo);
   APT_IMPLEMENT_RTTI_BASE(Foo, APT_Persistent);
     // list other bases, if any
   APT_IMPLEMENT_RTTI_END(Foo);

   Foo::Foo(int i, float f)
     : i_(i), f_(f)
   {
   }
   Foo::Foo()
     : i_(0), f_(0.0)
   {
   }
   void Foo::serialize(APT_Archive& ar, APT_UInt8)
   {
     ar || i_ || f_;
   }
*/

#define APT_DECLARE_ABSTRACT_PERSISTENT(T)                      \
public:                                                         \
  friend APT_Archive& operator|| (APT_Archive&, T*&);           \
  friend APT_Archive& operator|| (APT_Archive& ar, const T*& d) \
    { return ar || (T*&) d; }                                   \
  APT_DIRECTIONAL_POINTER_SERIALIZATION(T, friend)              \
private:                                                        \
  static void loadPointer(APT_Archive& ar, const APT_TypeInfo& ptrType, \
                          void* &rawPtr)                        \
    { APT_Persistent::loadPointer(ar, ptrType, rawPtr); }       \
  static void storePointer(APT_Archive& ar, APT_Persistent* ptr)\
    { APT_Persistent::storePointer(ar, ptr); }                  \
  void* wrapSerialize(void*, int, int&);                        \
  void serialize(APT_Archive&, APT_UInt8);                      \
  static int setupSerializeWrapper();                           \
  static int sWrapSetupDummy

#define APT_DECLARE_PERSISTENT(T)                               \
  APT_DECLARE_ABSTRACT_PERSISTENT(T);                           \
  virtual void dispatchSerialize(APT_Archive&);                 \
  virtual void reInit();                                        \
  static APT_Persistent* make();                                \
  static int sRegisterDummy
```

```
/* Note that the DECLARE macro declares serialize(), but the IMPLEMENT
   macro does not define it.  We depart from our usual convention
   (that anything declared in DECLARE macros is defined in
   corresponding IMPLEMENT macros) because we *really* want each
   derivation level to have serialize() (so that the generated
   wrapSerialize() will call the right function, not the inherited
   one), and we *really* want it to be private. */

#define APT_IMPLEMENT_ABSTRACT_PERSISTENT(T)                    \
  APT_IMPLEMENT_ABSTRACT_PERSISTENT_V(T,0)

#define APT_IMPLEMENT_ABSTRACT_PERSISTENT_V(T, CDV)             \
  APT_Archive& operator|| (APT_Archive& ar, T*& ptr)            \
  { if (ar.isLoading())                                         \
    { void* loadPtr;                                            \
      ptr = 0;                                                  \
      T::loadPointer(ar, APT_TYPE_INFO(T), loadPtr);            \
      ptr = (T*) loadPtr;                                       \
    }                                                           \
    else T::storePointer(ar, ptr);                              \
    return ar;                                                  \
  }                                                             \
  void* T::wrapSerialize(void* arch, int, int&)                 \
  {                                                             \
    if (!doThisBase( APT_TYPE_INFO(T) )) return 0;              \
    APT_Archive* ar = (APT_Archive*) arch;                      \
    APT_UInt8 version = CDV;                                    \
    *ar || version;                                             \
    T::serialize(*ar, version);                                 \
    return 0;                                                   \
  }                                                             \
  int T::setupSerializeWrapper()                                \
  { T::APT_sFunc = &T::wrapSerialize; return 0; }               \
  int T::sWrapSetupDummy = T::setupSerializeWrapper()

#define APT_IMPLEMENT_PERSISTENT(T)                             \
  APT_IMPLEMENT_PERSISTENT_V(T, 0)

#define APT_IMPLEMENT_PERSISTENT_V(T, CDV)                      \
  APT_IMPLEMENT_ABSTRACT_PERSISTENT_V(T, CDV);                  \
  void T::dispatchSerialize(APT_Archive& ar)                    \
    { int dummy=0; T::APT_baseTraverse(&ar, 1, dummy); }        \
  void T::reInit() { this->T::~T(); ::new(this) T; }            \
  APT_Persistent* T::make() { return new T; }                   \
  int T::sRegisterDummy =                                       \
    APT_Persistent::registerClass(APT_TYPE_INFO(T), T::make)
```

```
/* Alternate form for nested classes: T::N.  This grunge is needed
   because the explicit destructor call syntax gets into trouble
   if the ordinary IMPLEMENT macro is used with an class name argument
   of the form T::N. */
#define APT_IMPLEMENT_NESTED_PERSISTENT(T, N)                    \
  APT_IMPLEMENT_ABSTRACT_PERSISTENT_V(T::N, 0);                  \
  void T::N::dispatchSerialize(APT_Archive& ar)                  \
    { int dummy=0; T::N::APT_baseTraverse(&ar, 1, dummy); }      \
  void T::N::reInit() { this->T::N::~N(); ::new(this) T::N; }    \
  APT_Persistent* T::N::make() { return new T::N; }              \
  int T::N::sRegisterDummy =                                     \
    APT_Persistent::registerClass(APT_TYPE_INFO(T::N), T::N::make)

#endif // APT_PERSIST_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995, 1996 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_PROPLIST_H
#define APT_PROPLIST_H

#ifndef APT_STRING_H
#include <apt_util/string.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_ARCHIVE_H
#include <apt_util/archive.h>
#endif

#ifndef APT_IDENTIFIER_H
#include <apt_util/identifier.h>
#endif

class APT_ParseError;
class APT_Lexer;
class istream;

class APT_Property;

class APT_PropertyList
/* A property list is an ordered collection of name/value pairs, where
   the value can be:

     numeric (dfloat)
     string
     omitted (property name only)
     property list

   For the syntax for the textual representation of property lists,
   see doc/proplist_syntax.txt.

   Multiple properties with the same name are permitted.  The index
   order of properties matches the occurrence order of the properties
   in their textual representation.

   Values may appear alone in the list, without requiring a name=
   syntactic component (somewhat Lispish).  These properties have the
```

```
   empty string as their name.  However, there is an ambiguity:

     {foo=4, 5.5, bar}

   The first property foo=4 is well formed; the second property is
   well-formed (property name is empty string); the third property is
   problematic: is it a property named bar with no value, or an
   unnamed string property of value "bar"?  We resolve the ambiguity
   by saying it is a property named bar with no value (to be
   compatible with earlier versions of the property list abstraction
   where all property values required names).

   Note: the property list implementation is not (yet) particulary
   efficient.
*/
{
public:
  APT_PropertyList();
  APT_PropertyList(const APT_PropertyList&);
  APT_PropertyList& operator= (const APT_PropertyList&);
  ~APT_PropertyList();

  APT_PropertyList(const char*, APT_ParseError* err=0);
  APT_PropertyList(istream&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a property list (starting
               with the token '{').
               If the parse is successful, this APT_PropertyList
               object's state is replaced by the parsed property list.
     throws    APT_ParseError: trouble parsing the supplied input as a
                 property list.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
     requires  string arg non-null
  */

  int count() const;
  /*
     effect    Tells how many properties there are in this list.
  */

  void removeProperty(int index);
  /*
     effect    Removes the indicated property.
     requires  0 <= index < count()
  */
  void removeProperty(const APT_Identifier& pname);
  /*
     effect    Removes all properties with the indicated name.
               No effect if no properties match the indicated name.
```

```
   note      Property name matching is case-independent.
*/
void removeProperty(const APT_Identifier& pname, int index);
/*
   effect    Removes the indicated property.
   note      Property name matching is case-independent.
   requires  0 <= index < propertiesWithName(pname)
*/

bool hasProperty(const APT_Identifier& pname) const;
/*
   effect    Tells if one or more property with the given name
             exists.
   note      Property name matching is case-independent.
*/

int propertiesWithName(const APT_Identifier& pname) const;
/*
   effect    Tells how many properties there are with the given
             property name.
   note      Property name matching is case-independent.
*/

int lookupProperty(const APT_Identifier& pname, int which=0) const;
/*
   effect    Returns the index of a property with the given name.
             If more than one property has the given name, the which
             argument identifies which property's index should be
             returned.
             Returns -1 if hasProperty(pname) is false.
             The returned index value (if it is not -1) is suitable
             for calls to propertyName(), propertyKind(),
             propertyValueString(), and propertyValueList().
   note      Property name matching is case-independent.
   requires  pname non-null identifier
             0 <= index < propertiesWithName(pname)
*/

void replaceProperty(const APT_Property& prop, int which=0);
/*
   effect    Replaces the indicated property in this list with the
             argument.
   requires  0 <= which < propertiesWithName(prop.name())
*/

void addProperty(const APT_Property& prop, int where=-1);
/*
   effect    Adds the given property to this list.
             The where argument specifies the desired index of the
             added property.  -1 means add to the end of the list.
   note      Property names need not be unique.
```

```
              The argument property is copied.
   requires   where==-1  or 0 <= where < properties()+1
*/

const APT_Property& getProperty(const APT_Identifier& pname,
                                int which=0) const;
APT_Property& getProperty(const APT_Identifier& pname,
                          int which=0);
/*
   effect     Returns a property with the given name.
              If more than one property has the given name, the which
              argument identifies which property's index should be
              returned.
   note       Property name matching is case-independent.
   caution    The returned reference can become stale if this
              property list is modified.
   requires   hasProperty(pname) must be true.
              0 <= index < propertiesWithName(pname)
*/

const APT_Property& getProperty(int index) const;
APT_Property& getProperty(int index);
/*
   effect     Returns the property at the given position in the
              list.
   caution    The returned reference can become stale if this
              property list is modified.
   requires   0 <= index < count()
*/

const APT_Property& operator[] (int index) const {return getProperty(index);}
APT_Property& operator[] (int index) {return getProperty(index);}

void processIncludes(APT_String* errors, bool shellComments=true);
/*
   effect     For each top-level include="filename" property, removes
              the property, parses the indicated file as a property
              list, and adds the properties so found to the end of this
              property list.
              If any error occurs, an explanatory string is written
              to the errors arg.  The final state of this property
              list is unspecified if an error occurs.
   note       The shellComments flag controls whether #comments are
              recognized (true) or C/C++ comments are recognized.
*/

void parse(const char*, APT_ParseError* err=0);
void parse(istream&, APT_ParseError* err=0);
/*
   effect     Parses the input, expecting a property list (starting
              with the token '{').
```

```
              If the parse is successful, this APT_PropertyList
              object's state is replaced by the parsed property list.
   throws     APT_ParseError: trouble parsing the supplied input as a
                property list.  If the err argument is 0, then
                the error is thrown; if err is non-null, then the
                error object is copied into the object pointed to by
                err.
              If an error occurs, this APT_PropertyList object's
              state is unchanged.
   requires   string arg non-null
*/

void unparse(ostream&) const;
APT_String unparse() const;
/*
   effect     Prints or returns a parse()able representation of
              this property list.
*/

void prettyUnparse(ostream&, int indent=0) const;
// like unparse(), except tries to be more readable

APT_String prettyUnparse() const;
// implemented in terms of the ostream version

bool operator== (const APT_PropertyList&) const;
bool operator!= (const APT_PropertyList& rhs) const
  { return !(*this == rhs); }
/*
   effect     Tells whether the two property lists contain equal
              properties in the same order.
   note       Properties are compared via their operator==()
              function.
*/

bool hasSameProperties(const APT_PropertyList&) const;
/*
   effect     Tells whether the two property list contain equal
              properties, where order does not matter.
   note       This is currently quite inefficient.
              Properties are compared via their isSame() function.
*/

friend APT_Archive& operator|| (APT_Archive&, APT_PropertyList&);


/* old-style member functions; still supported, but less convenient
   than those above */

void addProperty(const char* pname, const char* pvalStr,
                 int where=-1);
```

```
void addProperty(const char* pname, const APT_String& pvalStr,
                 int where=-1);
void addProperty(const char* pname, const APT_PropertyList& pvalList,
                 int where=-1);
/*
   effect    Adds a property with the given name and value.
             Property values can be either strings or property
             lists.
             The where argument specifies the desired index of the
             added property.  -1 means add to the end of the list.
   note      Property names need not be unique.
             Property names are case-independent.
             Value strings may contain embedded whitespace.
   requires  pname non-null
             pname is an identifier (starts with letter or
             underscore, contains no whitespace).
             pvalStr non-null
             where==-1  or 0 <= where < properties()+1
*/

APT_String propertyName(int index) const;
/*
   effect    Returns the name of the indicated property.
   requires  0 <= index < properties()
*/

enum Kind
{
  eString,                    /* covers APT_Property::eEmpty,
                                 APT_Property::eString,
                                 APT_Property::eDFloat */
  ePropertyList
};
Kind propertyKind(int index) const;
/*
   effect    Identifies the kind of value held by the indicated
             property.
   note      Empty properties are treated by this function as
             empty eString.
   requires  0 <= index < properties()
*/

APT_String propertyValueString(int index) const;
/*
   effect    Returns the value of the indicated property.
   note      For empty properties, this returns the empty string.
   requires  0 <= index < properties()
             propertyKind(index) is eString
*/

const APT_PropertyList& propertyValueList(int index) const;
```

```
  /*
     effect    Returns the value of the indicated property.
     caution   The returned reference can become stale if this
               property list is modified.
     requires  0 <= index < properties()
               propertyKind(index) is ePropertyList
  */

  // end of old-style functions


  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
     effect    Parses using the supplied lexer object.
     note      If an error is detected, an attempt is made to advance
               the lexer past this property list.
  */

private:
  void copy(const APT_PropertyList&);

  int numProperties_;
  APT_Property* propList_;
};
APT_DIRECTIONAL_SERIALIZATION(APT_PropertyList);


class APT_Property
{
public:
  APT_Property();
  // eEmpty, empty name

  APT_Property(const APT_Identifier& name);
  // eEmpty

  // use empty name for nameless value property

  APT_Property(const APT_Identifier& name, const char* value);
  APT_Property(const APT_Identifier& name, const APT_String& value);
  // eString

  APT_Property(const APT_Identifier& name, APT_DFloat value);
  // eDFloat

  /* I won't add an APT_Int64 ctor overload, because this would cause lots
     of opportunities for ambiguity with the APT_DFloat ctor */

  APT_Property(const APT_Identifier& name, const APT_PropertyList& value);
  // eList
```

```
APT_Property(const char*, APT_ParseError* err=0);
APT_Property(istream&, APT_ParseError* err=0);
/*
   effect    Parses the input, expecting a property.
             If the parse is successful, this APT_Property object's
             state is replaced by the parsed property.
   throws    APT_ParseError: trouble parsing the supplied input as a
               property.  If the err argument is 0, then
               the error is thrown; if err is non-null, then the
               error object is copied into the object pointed to by
               err.
   requires  string arg non-null
*/

APT_Property(const APT_Property&);
APT_Property& operator= (const APT_Property&);
~APT_Property();


APT_Identifier name() const;
void setName(const APT_Identifier&);
/*
   effect    Controls the name of this property.
*/

enum Kind { eEmpty, eString, eDFloat, eList };

Kind kind() const;
void setKind(Kind k);
/*
   effect    Controls what kind of value this property has.
   note      If setKind() is used to set a property's type, the
             property's value is reset follows:
               eEmpty: no value
               eString: empty string
               eDFloat: 0.0
               eList: empty list
*/

static const APT_String &kindAsString(Kind k);
/*
   effect     Returns a string corresponding to the kind enumerator passed.
   requires   k be a valid Kind.
*/

void setValueEmpty();
/*
   effect    Sets this property to eEmpty.
*/

void setValueString(const APT_String&);
```

```
/*
   effect    Sets this property to eString with the given value.
*/
APT_String valueString() const;
/*
   effect    Returns this property's string value.
   requires  kind() == eString
*/

void setValueDFloat(APT_DFloat);
/*
   effect    Sets this property to eDFloat with the given value.
*/
APT_DFloat valueDFloat() const;
/*
   effect    Returns this property's floating point value.
   requires  kind() == eFloat
*/

void setValueInt64(APT_Int64);
/*
   effect    Sets this property to eDFloat with the given integer value.
*/

APT_Int64 valueInt64() const;
/*
   effect    Returns this property's integer value.
   note      If the setValueInt64() function was used, then the full
             64 bits integer precision is retained.
   requires  kind() == eFloat
*/

void setUnsignedInt64(bool);
/* effect    Sets variable, unsignedInt64_, to value of true or false.
             See comments for function declaration unsignedInt64() below.
   requires  kind() == eFloat and property's value is an integer
*/

void setSignedInt64(bool);
/* effect    Sets variable, signedInt64_, to value of true or false.
             See comments for function declaration signedInt64() below.
   requires  kind() == eFloat and property's value is a integer.
*/

bool unsignedInt64() const;
/*
    effect    Returns value of unsignedInt64_. Value returned will be true
              if property, based on its amount, is assumed to be a uint64.
              Can be used by client in int64 range checking to determine if
              a property's int64 value is out-of-range, i.e., if
              unsignedInt64() returns true and schema type is an int64,
```

```
              then property's value is out of range.
    requires  kind() == eFloat and property's value is an integer
*/

bool signedInt64() const;
/*
    effect    Returns value of signedInt64_. Value returned will be true
              if property, based on its amount, is assumed to be a int64.
              Can be used by client in uint64 range checking to determine
              if a property's uint64 value is out-of-range, i.e., if
              signedInt64() returns true and schema type is an uint64,
              then property's value is out of range.
    requires  kind() == eFloat and property's value is an integer
*/

bool isInteger() const { return isInteger_; }
/*
   effect    Tells whether this eFloat is really an integer.
   requires  kind() == eFloat
*/

void setValueList(const APT_PropertyList&);
/*
   effect    Sets this property to eList with the given value.
*/

const APT_PropertyList& valueList() const;
APT_PropertyList& valueList();
/*
   effect    Returns this property's list value.
   caution   The returned reference can become stale if this
             property (or the list it is part of) is modified.
   requires  kind() == eList
*/

APT_String valueAsString() const;
/*
   effect    Returns this property's value as follows:
               eEmpty: empty string
               eString: the string value
               eDFloat: the printed form of the dfloat value (not
                 sweating whether we print the full precision)
               eList: the unparse() of the list value
*/

inline bool stringIsExact( ) const { return (valueString_ != ""); }
/*
   effect   True if valueAsString( ) returns an exact representation;
            that is, if the value was originally stored as a string
            (e.g. when parsing from an istream) and is not being
            generated by a conversion from another type
```

```
  */

  void parse(const char*, APT_ParseError* err=0);
  void parse(istream&, APT_ParseError* err=0);
  /*
     effect    Parses the input, expecting a property.
               If the parse is successful, this APT_Property object's
               object's state is replaced by the parsed property name
               and value (if any).
     throws    APT_ParseError: trouble parsing the supplied input as a
                 property.  If the err argument is 0, then
                 the error is thrown; if err is non-null, then the
                 error object is copied into the object pointed to by
                 err.
               If an error occurs, this APT_Property object's state is
               unchanged.
     requires  string arg non-null
  */

  void unparse(ostream&) const;
  APT_String unparse() const;
  /*
     effect    Prints or returns a parse()able representation of
               this property.
  */

  bool operator== (const APT_Property&) const;
  bool operator!= (const APT_Property& rhs) const { return !(*this == rhs); }
  /*
     effect    Equality comparison based on same name, same type, and
               same value.
     note      In the case of eList properties, the list values are
               compared via their operator==() function.
  */

  bool isSame(const APT_Property&) const;
  /*
     effect    Equality comparison based on same name, same type, and
               same value.
     note      In the case of eList properties, the list values are
               compared via their hasSameProperties() function.
  */

  friend APT_Archive& operator|| (APT_Archive&, APT_Property&);

private:
  friend class APT_PropertyList;

  APT_String prettyUnparse() const;
  // requires: not eList
```

```
  void parse_(APT_Lexer&, APT_ParseError* err);
  /*
     effect    Parses using the supplied lexer object.
     note      If an error is detected, the offending token is pushed
               back; *no* attempt is made to advance the lexer past
               this property (this nastiness is left up to the caller).
  */

  APT_Identifier name_;
  Kind kind_;

  APT_DFloat valueDFloat_;
  APT_Int64 valueInt64_;
  bool unsignedInt64_;
  bool signedInt64_;
  bool isInteger_;              // setValueInt64() was called
  APT_PropertyList* valueList_;
  APT_String valueString_; // value sometimes kept as string too
};
APT_DIRECTIONAL_SERIALIZATION(APT_Property);

#endif // APT_PROPLIST_H
```

```
// -*-Mode: C++-*-
//#
//#  Copyright (C) 1995 Torrent Systems, Inc. All Rights Reserved
//#
//#  $Id: random.h,v 1.6 2000/06/22 16:37:23 linda Exp $

#ifndef APT_RANDOM_H
#define APT_RANDOM_H

#ifndef APT_PERSIST_H
#include <apt_util/persist.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

/*
   OVERVIEW

      This is essentially the BSD random number generator with
      a C++ style interface. The BSD copyright notice, and the
      comments for the original C code, appear at the end of
      this file.

      This random number generator is provided as a convenience
      to Orchestrate users, who are solely responsible for evaluating
      its usefulness in any particular application. The reasons for
      providing this code are:

      - Orchestrate internally requires random numbers. Since system-
        vendor-provided generators are notoriously unreliable (though
        this seems to be changing...) we include one that works for
        all systems.

      - The characteristics of a random number generator are often
        so important that a cautious user will want to see the source
        code to the generator. By providing the BSD generator we
        can freely offer source code.

      A single generator can be used as a source for integers and
      floats. The sequence of delivered numbers is repeatable given
      the same constructor arguments and identical sequence of calls.

      Note that applications that generate random numbers in parallel
      need to consider whether or not the same sequence should be
      delivered to all instances of a parallel operator, and manage
      the seed accordingly.
*/
```

```
class APT_RandomNumberGenerator : public APT_Persistent
{
    APT_DECLARE_RTTI(APT_RandomNumberGenerator);
    APT_DECLARE_PERSISTENT(APT_RandomNumberGenerator);

  public:

    APT_RandomNumberGenerator(APT_Int32 period,
                              APT_Int32 seed);
    /*
       effects  Initialize a random number generator of the specified
                period using the using the specified seed. "Period" is
                a number from 0 to 4 that provides a strictly relative
                indicator of periodicity; higher settings result in
                longer periods than lower periods. No absolute statement
                as periodicity should be inferred.

       requires 0 <= period <= 4

                seed > 0
    */


    APT_RandomNumberGenerator();
    /*
       effects  Create an uninitialized random number generator. The
                period and seed can be set with setPeriod() and setSeed().
                The default seed (1) and/or period (3) are used if a
                random number is requested without setSeed() and/or
                setPeriod() having been called.
    */


    // Compiler supplied copy ctor, assignment and dtor can't be used.
    APT_RandomNumberGenerator(const APT_RandomNumberGenerator & rhs);
    APT_RandomNumberGenerator & operator=(const APT_RandomNumberGenerator & rhs);
    ~APT_RandomNumberGenerator();


    void setSeed(APT_Int32 seed);
    /*
       effects  Sets the seed to be used to start the generator.

       requires The seed must not already be set (seed() == -1).
                seed > 0
    */


    void setPeriod(APT_Int32 period);
    /*
       effects  Sets the period of the generator.
```

```
      requires The period must not already be set (period() == -1).
               0 <= period <= 4
   */


   void setLimit(APT_Int32 limit);
   void setLimitInt64(APT_Int64 limit);
   void setLimitUInt64(APT_UInt64 limit);
   /*
      effects  Sets the upper bound on numbers returned by calls to
               nextInteger(), nextInt64(), or nextUInt64(). The number
               will be in the range:

                  0 <= number < limit

      requires limit > 0
   */


   APT_Int32 nextInteger();
   APT_Int64 nextInt64();
   APT_UInt64 nextUInt64();
   /*
      effects  Generates the "next" random number and uses it to return
               a number in the range defined by setLimit(), setLimtInt64(),
               or setLimitUInt64.

      requires setLimit(), setLimitInt64(), or setLimitUInt64() to have
      been called to establish the range.
   */


   APT_SFloat nextSFloat();
   /*
      effects  Generates a random number of type APT_SFloat in the range:

               0 <= number < 1.0
   */


   APT_DFloat nextDFloat();
   /*
      effects  Generates a random number of type APT_DFloat in the range:

               0 <= number < 1.0

      notes    This function takes about twice as much time to execute as
               nextSFloat because it must generate twice as many random bits.
   */


   APT_Int32 seed() const;
```

```
   /*
      effects  Returns the seed that was used to start the random
               number generator. If the seed has not yet been set
               (either via the constructor or setSeed()) then -1
               will be returned.
   */


   APT_Int32 period() const;
   /*
      effects  Returns the period of the generator. If the period has
               not yet been set (either via the constructor or setPeriod())
               then -1 will be returned.
   */


   APT_Int32 limit() const;
   /*
      effects  Returns the value passed to setLimit(). If setLimit
               was never called, the value returned is -1.
   */


   APT_Int64 limitInt64() const;
   /*
      effects  Returns the value passed to setLimitInt64(). If setLimitInt64
               was never called, the value returned is -1.
   */

   APT_UInt64 limitUInt64() const;
   /*
      effects  Returns the value passed to setLimitUInt64(). If setLimitUInt64
               was never called, the value returned is -1.
   */


 private:

   APT_Int32 seed_;
   APT_Int32 period_;
   APT_Int32 limit_;
   APT_Int64 limitInt64_;
   APT_UInt64 limitUInt64_;

   // Other stuff needed by the implementation

   void initialize();
   APT_Int32 nextRandom();

   // These are standalone routine to enable them to be
   // independently tested.
```

```
    static
    APT_SFloat sfloatFromRandom(APT_Int32 bits);
    /*
       effects  Use the 31 random bits supplied to generate
                an APT_SFloat in the range 0 < x < 1.

       note     Not all of the bits will necessarily be used.
    */


    static
    APT_DFloat dfloatFromRandom(APT_Int32 hiBits,
                                APT_Int32 loBits);
    /*
       effects  Use the 62 random bits supplied to generate
                an APT_DFloat in the range 0 < x < 1.

       note     Not all of the bits will necessarily be used.
    */

    // Expose private interfaces for unit testing.
    friend class APT_RandomNumberGenerator_UT;

    APT_Int32 stateWords_;
    APT_Int32 * state_;
    APT_Int32 * endptr_;
    APT_Int32 * fptr_;
    APT_Int32 * rptr_;
    APT_Int32 separation_;
};

/*
 * Copyright (c) 1983 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by the University of
 *      California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
```

```
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */



/*
 * An improved random number generation package.  In addition to the standard
 * rand()/srand() like interface, this package also has a special state info
 * interface.  The initstate() routine is called with a seed, an array of
 * bytes, and a count of how many bytes are being passed in; this array is
 * then initialized to contain information for random number generation with
 * that much state information.  Good sizes for the amount of state
 * information are 32, 64, 128, and 256 bytes.  The state can be switched by
 * calling the setstate() routine with the same array as was initiallized
 * with initstate().  By default, the package runs with 128 bytes of state
 * information and generates far better random numbers than a linear
 * congruential generator.  If the amount of state information is less than
 * 32 bytes, a simple linear congruential R.N.G. is used.
 *
 * Internally, the state information is treated as an array of longs; the
 * zeroeth element of the array is the type of R.N.G. being used (small
 * integer); the remainder of the array is the state information for the
 * R.N.G.  Thus, 32 bytes of state information will give 7 longs worth of
 * state information, which will allow a degree seven polynomial.  (Note:
 * the zeroeth word of state information also has some other information
 * stored in it -- see setstate() for details).
 *
 * The random number generation technique is a linear feedback shift register
 * approach, employing trinomials (since there are fewer terms to sum up that
 * way).  In this approach, the least significant bit of all the numbers in
 * the state table will act as a linear feedback shift register, and will
 * have period 2^deg - 1 (where deg is the degree of the polynomial being
 * used, assuming that the polynomial is irreducible and primitive).  The
 * higher order bits will have longer periods, since their values are also
 * influenced by pseudo-random carries out of the lower bits.  The total
 * period of the generator is approximately deg*(2**deg - 1); thus doubling
 * the amount of state information has a vast influence on the period of the
 * generator.  Note: the deg*(2**deg - 1) is an approximation only good for
 * large deg, when the period of the shift register is the dominant factor.
 * With deg equal to seven, the period is actually much longer than the
 * 7*(2**7 - 1) predicted by this formula.
 */

#endif
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_RTTI_H
#define APT_RTTI_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

/* Simple RTTI package for implementing checked casts.  This scheme is
   inspired by (but is somewhat improved over) the approach outlined
   in _The C++ Programming Language_ by Stroustrup, 2d edition.  See
   section 13.5 "Run-time Type Information".

   Our RTTI package has the following desirable attributes:

   - no common base class required
   - support for multiple inheritance
   - support for virtual base classes
   - can downcast from virtual base class
   - supports checked casts from/to any base pointer or derived pointer
     (see notes 1 and 2)

   Note 1: Classes to/from which you would like to do checked casts
   must use APT_{DECLARE,IMPLEMENT}_RTTI macros as outlined below.

   Note 2: A checked cast to an ambiguous base class will yield the
   base class reached first by the series of APT_IMPLEMENT_RTTI_BASE
   definitions for the joining class.

   Usage example:

   foo.h:
     class Foo : public B1,     // classes B1 and B2 should have RTTI
                 public B2      // macros to participate in checked casts
     {
       APT_DECLARE_RTTI(Foo);
     public:
       ...
     };

   foo.C:
     APT_IMPLEMENT_RTTI_BEGIN(Foo);
     APT_IMPLEMENT_RTTI_BASE(Foo, B1);
     APT_IMPLEMENT_RTTI_BASE(Foo, B2);
     APT_IMPLEMENT_RTTI_END(Foo);
```

```
    ...

   Note that the APT_IMPLEMENT_RTTI_{BEGIN,BASE,END} macros must appear
   together as illustrated above.  This is because the BEGIN macro
   starts a function definition, the BASE macros (if any) continue the
   function definition, and the END macro ends the function definition.

   For cases where the class being defined has no bases or one base,
   the APT_IMPLEMENT_RTTI_NOBASE and APT_IMPLEMENT_RTTI_ONEBASE macros can
   be used, respectively.

   Checked casts can be performed using the APT_PTR_CAST(T, p) macro,
   which converts the supplied pointer p to a T*, yielding either the
   resulting pointer or 0.  0 is returned if the p pointer does not
   refer to a T or something derived from T.

   B1* bp = get_a_b1();
   Foo* fp = APT_PTR_CAST(Foo, bp);
   if (fp) { ... }

   Eventually C++ will support built-in rtti.  However, this RTTI
   package will be retained because it provides support beyond
   built-in rtti (such as internal hooks required for our persistence
   package).
*/

/* All of the following macros tolerate trailing semi-colons in their
   usage; some require a trailing semi-colon. */

/* Macros for defining RTTI information.  These are probably going to
   be trouble for multi-parameter template classes; multi-parameter
   temlate versions of these macros will be provided if needed.  See
   smr. */

#define APT_DECLARE_RTTI(Class)                                         \
public:                                                                 \
  virtual const APT_TypeInfo& APT_dynType() const;                      \
  virtual void* APT_castTo(const APT_TypeInfo&) const;                  \
  virtual const void* APT_objectBase() const;                           \
  static const APT_TypeInfo& APT_staticType();                          \
protected:                                                              \
  void* APT_baseTraverse(void* arg, int op, int& done) const;           \
  static void* (Class::* APT_sFunc)(void*, int, int&);                  \
private:                                                                \
  static APT_TypeInfo APT_sInfo

#define APT_IMPLEMENT_RTTI_BEGIN(C)                                     \
  const APT_TypeInfo& C::APT_dynType() const { return C::APT_sInfo; }   \
```

```
  const APT_TypeInfo& C::APT_staticType() { return C::APT_sInfo; }      \
  void* C::APT_castTo(const APT_TypeInfo& type) const                   \
  {                                                                     \
    int dummy=0;                                                        \
    return APT_baseTraverse((void*) &type, 0, dummy);                   \
  }                                                                     \
  const void* C::APT_objectBase() const                                 \
  {                                                                     \
    return this;                                                        \
  }                                                                     \
  void* C::APT_baseTraverse(void* arg, int op, int& done) const         \
  {                                                                     \
    if (op == 0 && ((const APT_TypeInfo*) arg == &C::APT_sInfo))        \
    { done=1;                                                           \
      return (void*) this;                                              \
    }                                                                   \
    void* answer=0

#define APT_IMPLEMENT_RTTI_BASE(Class, Base)                            \
    answer = Base::APT_baseTraverse(arg, op, done);                     \
    if (done) return answer

#define APT_IMPLEMENT_RTTI_END(C)                                       \
    if (op != 0 && C::APT_sFunc)                                        \
    answer = (((C*) this)->*C::APT_sFunc)(arg, op, done);               \
    if (done) return answer;                                            \
    return 0;                                                           \
  }                                                                     \
  void* (C::* C::APT_sFunc)(void*, int, int&);                          \
  APT_TypeInfo C::APT_sInfo(#C)
/* note: for template classes with RTTI, the string-ified Class name
   will be incorrect: the <T> portion of the template name will not be
   substituted for the actual template argument.  See smr. */

// convenient forms for classes with no RTTI base class
#define APT_IMPLEMENT_RTTI_NOBASE(Class)                                \
  APT_IMPLEMENT_RTTI_BEGIN(Class);                                      \
  APT_IMPLEMENT_RTTI_END(Class)

// convenient forms for classes with one RTTI base class
#define APT_IMPLEMENT_RTTI_ONEBASE(Class, Base)                         \
  APT_IMPLEMENT_RTTI_BEGIN(Class);                                      \
  APT_IMPLEMENT_RTTI_BASE(Class, Base);                                 \
  APT_IMPLEMENT_RTTI_END(Class)


// macros for using RTTI information
```

```
#define APT_PTR_CAST(T, p)      \
  ( (T*) ((p) ? (p)->APT_castTo(T::APT_staticType()) : 0) )
#define APT_TYPE_INFO(T) T::APT_staticType()
#define APT_STATIC_TYPE(o) (o).APT_staticType()
#define APT_DYNAMIC_TYPE(o) (o).APT_dynType()
#define APT_NAME_FROM_TYPE(t) (t).name()


class APT_TypeInfo
{
public:
  APT_TypeInfo(const char* className) : className_(className) {}
  const char* name() const { return className_; }

  friend bool operator== (const APT_TypeInfo& lhs, const APT_TypeInfo& rhs)
  { return &lhs == &rhs; }
  friend bool operator!= (const APT_TypeInfo& lhs, const APT_TypeInfo& rhs)
  { return &lhs != &rhs; }

private:
  // prohibit copying
  APT_TypeInfo(const APT_TypeInfo&);
  APT_TypeInfo& operator= (const APT_TypeInfo&);

  const char* className_;
};

#endif // APT_RTTI_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1995,1998 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_STRING_H
#define APT_STRING_H

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_STATUS_H
#include <apt_util/status.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

#include <string.h>

class APT_Archive;
class istream;
class ostream;

class APT_String
/* APT_String has the following public characteristics:

     - value semantics
     - not null terminated (explicit length)
     - can contain null chars
     - can be fixed-length or variable-length
     - ASCII character set

   In addition to being useful as a general-purpose string class,
   APT_String is also the C++ field value type corresponding to the
   "string" schema field type.

   In the member function specs below, a "free" string instance is one
   that is constructed and manipulated by the client using ordinary
   C++ means; a "bound" string instance is one that has been created
   by the framework to represent the value of a record's string field.
   Bound string instances may be accessed via
   APT_InputAccessorToString (const operations only) and
   APT_OutputAccessorToString (const and non-const operations).

   In the function signatures below, a char* paired with a APT_Int32
   indicates a character string with an explicit length, where the
```

```
   string can contain embedded nulls and is not necessarily
   null-terminated (the explicit length does not include a terminating
   null character, if one is present).

   By contrast, a char* appearing alone (without a APT_Int32 arg)
   signifies a null-terminated character string.

   No derived classes are anticipated.
*/
{
public:
  // ctors, dtor

  APT_String();
  APT_String(const char*);
  APT_String(const char*, APT_Int32 len);
  APT_String(char);

  APT_String(const APT_String&);
  ~APT_String();


  // assignment

  APT_String& operator= (const APT_String& rhs);
  /*
     effect    If isFixedLength() is true, copies min(length(),
               rhs.length()) characters from rhs into this string.
               If less than length() characters are copied, the
               balance of the length() is right-filled with pad
               characters.
               If isBoundedLength() is true, silently clamps the copied
               length according to the bound.
               Only the string value is copied; attributes such as
               fixed-length, pad char, etc. are not copied.
     note      For variable-length strings, it is not necessary to call
               setLength() before calling this function.
  */
  APT_String& operator= (const char* rhs);
  APT_String& operator= (char c);

  void assignFrom(const APT_String& str);
  void assignFrom(const char* str, APT_Int32 len);
  void assignFrom(const char* str);
  void assignFrom(char c);
  /*
     effect    Like operator=, except the first overload allows an
               explicit buffer and length to be specified.
               Only the string value is copied; attributes such as
               fixed-length, pad char, etc. are not copied.
  */
```

```
// pad character (for fixed-length strings)

inline char padChar() const;
void setPadChar(char c);
/*
   effect    Specifies the pad character used for fixed length
             strings.  The default pad character is ASCII NUL,
             (char) 0.
   note      The string's present value is not changed by
             setPadChar(); only future assignments are affected.
*/


// length

inline APT_Int32 length() const;
/*
   effect    Returns the length of this string.
   note      For a fixed-length string, length() always return
             the string's fixed length, regardless of the string's
             value.
*/
bool isEmpty() const;
/*
   effect    Tells if this string is empty.  A variable-length
             string is empty when its length() is zero; a
             fixed-length string is empty when all of its characters
             are the padChar(),
*/

void setLength(APT_Int32 len);
/*
   effect    Sets the length of this string.
             If the length is reduced, the string's right-most
             characters are dropped; if the length is increased, the
             string is right-filled with pad characters.
             If isBoundedLength() is true, silently clamps the argument
             length according to the bound.
   requires  isVariableLength() must be true.
*/


// access to string character array representation

char* nonTerminatedContent();
const char* nonTerminatedContent() const;
/*
   effect    Returns a pointer to this string's contents.
             Returns 0 if this string's length() is zero.
   note      The contents returned by this function are *not*
             necessarily null-terminated.
```

```
              The caller may access up to length() characters using
              the returned pointer.
              The caller must not retain the returned pointer, as the
              underlying storage can be invalidated by many things,
              including (but not least) non-const operations on this
              string instance.
  */


  const char* terminatedContent() const;
  operator const char* () const;
  /* Access string as a C-compatible null-terminated string.  In
     addition to being null-terminated, the string value can contain
     embedded nulls; consult length() to find the length of the
     string excluding the terminating null.
     Returns a pointer to a null char if this string's length() is
     zero.
     The terminating null character is not part of this string's
     length().
     (Caution: the returned pointer becomes invalid when this string
     is destroyed or assigned to). */


  // char extract (subscripting)

  char operator[] (APT_Int32 index) const;
#ifdef APT_INT32_IS_NOT_INT
  char operator[] (int index) const;
#endif
  char operator[] (unsigned int index) const;

  /*
     effect    Retrieves the character at the indicated (0-based)
               offset.
     note      If index==length(), 0 is returned.  That is, this
               operator provides null-terminated semantics.
     require   0 <= index <= length().  The [] operators do not
               actually check this requirement, so client beware.
  */

  char getChar(APT_Int32 index) const;
  /*
     effect    Like operator[], except the range check implied by the
               requires clause is guaranteed to be performed.
     require   0 <= index <= length()
  */


  // substring by character position

  APT_String substring(APT_Int32 offset, APT_Int32 len) const;
  /*
```

```
   effect    Returns the indicated substring of this string.
             The offset arg is 0-based.
   note      Excessive args are silently clipped, so for
             example:
                APT_String s = "hello"
               APT_ASSERT(s.substring(2, 10) == "llo");
               APT_ASSERT(s.substring(5, 1) == "");
             The returned string is always variable-length.
   requires  0 <= offset
*/


// substring by content

enum CaseMatch { eCaseSensitive=0, eCaseInsensitive };
enum Direction { eBeginning=0, eEnd };

int occurrences(const APT_String& substr, CaseMatch c=eCaseSensitive) const;
int occurrences(const char* substr, APT_Int32 len,
                CaseMatch c=eCaseSensitive) const;
int occurrences(const char* substr, CaseMatch c=eCaseSensitive) const;
int occurrences(char ch, CaseMatch=eCaseSensitive) const;
/*
   effect    Tells how many times the given substring occurs within
             this string.
             If the argument is the empty string, then 0 is
             returned.
             Examples:
               APT_String s = "how now, brown cow";
               s.occurrences("ow") ==> 4
               s = "mmm";
               s.occurrences("mm") ==> 2
               s.occurrences('x') ==> 0
*/


APT_Int32 offsetOfSubstring(const APT_String& substr,
                            CaseMatch c=eCaseSensitive,
                            Direction d=eBeginning, int which=0) const;
APT_Int32 offsetOfSubstring(const char* substr, APT_Int32 sublen,
                            CaseMatch c=eCaseSensitive,
                            Direction d=eBeginning, int which=0) const;
APT_Int32 offsetOfSubstring(const char* substr,
                            CaseMatch c=eCaseSensitive,
                            Direction d=eBeginning, int which=0) const;
APT_Int32 offsetOfSubstring(char ch,
                            CaseMatch c=eCaseSensitive,
                            Direction d=eBeginning, int which=0) const;
/*
   effect    Returns the position of the first (or last) substring
             of this string matching the entire argument string.  If
             no match is found, -1 is returned.
```

```
              The which argument determins which match is returned.
              By default, the first match (numbered as 0) is
              returned.
              If the argument is the empty string, then the beginning
              (0) or end (length) offset of this string is returned.
              Examples:
                APT_String s = "a talking frog is way cool"
                s.offsetOfSubstring("talk") ==> 2
                s.offsetOfSubstring("toad") ==> -1
                s.offsetOfSubstring("FROG", APT_String::eCaseInsensitive)
                   ==> 10
                s.offsetOfSubstring("g", APT_String::eCaseSensitive) ==> 8
                s.offsetOfSubstring('g', APT_String::eCaseSensitive,
                                    APT_String::eEnd) ==> 13
                s.offsetOfSubstring('g', APT_String::eCaseSensitive,
                                    APT_String::eBeginning, 1) ==> 13
 */

 APT_String substring(const APT_String& substr,
                      CaseMatch c,
                      Direction d=eBeginning, int which=0) const;
 APT_String substring(const char* substr, APT_Int32 sublen,
                      CaseMatch c,
                      Direction d=eBeginning, int which=0) const;
 APT_String substring(const char* substr,
                      CaseMatch c,
                      Direction d=eBeginning, int which=0) const;
 APT_String substring(char ch,
                      CaseMatch c,
                      Direction d=eBeginning, int which=0) const;
 /*
    effect    Returns the substring whose starting offset is given by
              offsetOfSubstring(), and whose length spans to the
              end of this string.
              If offsetOfSubstring() is -1, returns the empty string.
              Examples:
                APT_String s = "a talking frog is way cool"
                s.substring("way") ==> "way cool"
                s.substring('c', APT_String::eCaseSensitive,
                            APT_String::eEnd) ==> "cool"
                s.substring('g', APT_String::eCaseSensitive,
                            APT_String::eBeginning, 1) ==> "g is way cool"
    note      The returned substring continues to the end of this
              string regardless of the specified Direction.
              The returned string is always variable-length.
              The CaseMatch parameter is not defaulted, because
              doing so would cause potential ambiguity between the
              second overloads and substring(0, 0).
 */


 // equality comparison
```

```
bool equals(const APT_String& str, CaseMatch c=eCaseSensitive) const;
bool equals(const char* str, APT_Int32 len,
            CaseMatch c=eCaseSensitive) const;
bool equals(const char* str, CaseMatch c=eCaseSensitive) const;
bool equals(char ch, CaseMatch c=eCaseSensitive) const;

friend bool operator== (const APT_String& lhs, const APT_String& rhs);
friend bool operator!= (const APT_String& lhs, const APT_String& rhs);

friend bool operator== (const char* lhs, const APT_String& rhs);
friend bool operator!= (const char* lhs, const APT_String& rhs);

friend bool operator== (const APT_String& lhs, const char* rhs);
friend bool operator!= (const APT_String& lhs, const char* rhs);

friend bool operator== (char lhs, const APT_String& rhs);
friend bool operator!= (char lhs, const APT_String& rhs);

friend bool operator== (const APT_String& lhs, char rhs);
friend bool operator!= (const APT_String& lhs, char rhs);


// ordered comparison

enum CompareResult { eLessThan=-1, eEqual=0, eGreaterThan=1 };

CompareResult compare(const APT_String& str,
                      CaseMatch c=eCaseSensitive) const;
CompareResult compare(const char* str, APT_Int32 len,
                      CaseMatch c=eCaseSensitive) const;
CompareResult compare(const char* str,
                      CaseMatch c=eCaseSensitive) const;
/*
   effect    Returns a code indicating whether this string is less
             than, equal to, or greater than the argument string.
   note      Comparisons are based on the ASCII collating sequence.
*/


// append

void append(const APT_String& str, Direction d=eEnd);
void append(const char*, APT_Int32, Direction d=eEnd);
void append(const char* str, Direction d=eEnd);
void append(char c, Direction d=eEnd);
/*
   effect    Appends the argument string to the end (or beginning)
             of this string.
             Examples:
               APT_String s = "middle";
               s.append(", end") ==> "middle, end"
```

```
                s.append("start, ", APT_String::eBeginning) ==>
                  "start, middle, end"
   note      If this string is fixed-length and the Direction is
             eEnd, there is no effect.
             See also APT_StringAccum.
*/

void prepend(const APT_String& str);
void prepend(const char*, APT_Int32);
void prepend(const char* str);
void prepend(char c);
// append() with Direction hardwired to eBeginning

APT_String& operator+= (const APT_String& str);
APT_String& operator+= (const char* str);
APT_String& operator+= (char c);
// sugar for append()

friend APT_String operator+ (const APT_String&, const APT_String&);
friend APT_String operator+ (const APT_String&, const char*);
friend APT_String operator+ (const char*, const APT_String&);
friend APT_String operator+ (const APT_String&, char);
friend APT_String operator+ (char, const APT_String&);
/* similar to append(), except constructs a fresh variable length
   string instance */


// case alteration

void toUpper();
void toLower();
/*
   effect    Mutates string to all upper or lower case.
             Non-alphabetic characters are not altered.
   note      These operations are based on the ASCII character set.
*/


// substring editing

void replace(APT_Int32 offset, APT_Int32 span, const APT_String& src);
void replace(APT_Int32 offset, APT_Int32 span,
             const char* src, APT_Int32 srcLen);
void replace(APT_Int32 offset, APT_Int32 span, const char* src);
void replace(APT_Int32 offset, APT_Int32 span, char c);
/*
   effect    Cuts out the substring indicated by offset/span, and
             pastes in the src string arg.
             Examples:
               APT_String s = "a small brown dog";
               s.replace(0, 1, "two")  ==> "two small brown dog"
               s.replace(0, 9, "a big") ==> "a big brown dog"
```

```
   requires  0 <= offset
             0 <= span
             offset + span <= length()
*/


// padding

void trimPadding(Direction dir=eEnd, Direction justify=eBeginning);
void trimPadding(char padChar,
                 Direction dir=eEnd, Direction justify=eBeginning);
/*
   effect    Chops off any trailing (or leading, if dir==eBeginning)
             characters equal to the supplied pad char.  If a pad
             char is not supplied, then padChar() is used.
             For fixed-length strings, the resulting string value is
             left-justified (or right-justified, if justify==eEnd),
             and the string's padChar() used to perform any new
             filling.
             Examples:
               APT_String var = "  a word or two   ";
               var.setPadChar(' ');
               var.trimPadding();
               APT_ASSERT(var == "  a word or two");

               var = "  a word or two   ";
               var.trimPadding(APT_String::eBeginning);
               APT_ASSERT(var == "a word or two   ");

               APT_String fix;
               fix.setPadChar(' ');
               fix.setFixedLength(10);
               fix = "  hello   ";
               fix.trimPadding();
               APT_ASSERT(fix == "  hello   "); // no change
               fix.trimPadding(APT_String::eBeginning);
               APT_ASSERT(fix == "hello     ");
               fix.trimPadding(APT_String::eEnd, APT_String::eEnd);
               APT_ASSERT(fix == "     hello");
*/


/* converions to built-in C/C++ types; conversions to other types
   handled via those types' member functions */

APT_Int32 asInteger(APT_Int32 failVal=0, APT_Status* flag=0) const;
/*
   effect    Parses this string as a signed decimal integer value.
             If an error occurs, the indicated failVal is returned;
             the optional flag is set to APT_StatusOk or
             APT_StatusFailed if provided.
   note      Leading whitespace is ignored; trailing cruft is not
```

```
              diagnosed.  For example:
                APT_String s = "  5 golden rings";
                APT_Status stat;
                APT_Int32 numRings = s.asInteger(0, &stat);
                APT_ASSERT(numRings == 5);
                APT_ASSERT(stat == APT_StatusOk);
*/

APT_DFloat asFloat(APT_DFloat failVal=0, APT_Status* flag=0) const;
/*
   effect    Parses integer value via strtod.
             If an error occurs, the indicated failVal is returned;
             the optional flag is set to APT_StatusOk or
             APT_StatusFailed if provided.
   note      Like asInteger(), leading whitespace is ignored;
             trailing cruft is not diagnosed.
*/


// hash

APT_UInt32 hash(CaseMatch c=eCaseSensitive) const;
/*
   effect    Computes a hash value for this string.
*/


// iostream support

friend ostream& operator<< (ostream&, const APT_String&);
/*
   effect    Prints all length() characters of the given string
             instance.
*/
friend istream& operator>> (istream&, APT_String&);
/*
   effect    Reads the next string from the given istream into the
             given string instance.  Leading whitespace is consumed
             and discarded; a string is read until more whitespace
             (or EOF) is encountered; trailing whitespace is not
             consumed.
*/


// Orchestrate persistence

friend APT_Archive& operator|| (APT_Archive&, APT_String&);
/*
   effect    Stores/loads the given string's content and length.
             Other attributes (fixed/var/bounded length, pad char,
             etc.) are not serialized.
*/
```

```
// length options

/* For free instances, isVariableLength() is true unless set
   otherwise by the client.  For bound instances, the length mode of
   the string instance is determined by the field's type parameters,
   and cannot be changed by the client. */

bool isFixedLength() const { return bits_.isFixedLength_ ? true : false; }
/*
   effect    Tells if this string is fixed length.  If so,
             length() will always report the same value.
*/
bool isBoundedLength() const { return bits_.isBoundedLength_ ? true : false; }
/*
   effect    Tells if this string is bounded length.
   note      Do not confuse "bounded length" with the notion of a
             bound instance.
*/
bool isVariableLength() const { return !bits_.isFixedLength_; }
/*
   effect    Tells if this string is variable-length.  Free
             instances are variable length unless set otherwise by
             the client.
*/

void setVariableLength();
/*
   effect    Sets this string instance to variable length.  The
             existing string value is replaced by the empty string.
   requires  isBound() must be false.
*/
void setFixedLength(APT_Int32 len);
/*
   effect    Sets this string instance to fixed length.  The
             existing string value is replaced by the empty string
             (padded with the current padChar()).
   requires  isBound() must be false.
             0 < len <= 0xfffff
*/
void setBoundedLength(APT_Int32 len);
/*
   effect    Sets this string instance to bounded length.  The
             existing string value is replaced by the empty string.
   requires  isBound() must be false.
             0 < len <= 0xfffff
*/


// cruft...
```

```
  bool isBound() const { return bits_.isBound_ ? true : false; }
  /*
     effect    Tells if this string instance is "bound" to a concrete
               record field whose processing is governed by the
               framework.
     note      Bound instances have certain restrictions, notably that
               you cannot change the length mode on bound strings.
  */

  void adopt(char* str, APT_Int32 len);
  void adopt(char* str);
  /*
     effect    Like assignFrom(), except that this string object takes
               ownership of the passed pointer.
     note      For the APT_String& overload, the argument string
               becomes empty.
     requires  For the char* overloads:
                 str must be delete[]-able.
                 str[len] must be a null char.
                 After calling this function, the client must not
                 delete the string that was passed to this function.
  */

  void bind(const char* content, APT_Int32 len);
  /*
     effect    Like assignFrom(), except that under certain
               circumstances the data are not copied and are instead
               referenced.
     requires  The content must remain valid until either this string
               field is destroyed, or a bind(), adopt(), op=, or
               assignFrom() is performed.
  */

  // back-compatible function names
  inline char* content();
  inline const char* content() const;
  const char* data() const;
  bool isEqualCI(const APT_String& str) const;
  bool isEqualCI(const char* str, APT_Int32 len) const;
  bool isEqualCI(const char* str) const;

private:
  friend class APT_StringDescriptor;
  friend class APT_String_UT;

  static char sEmptyStr;        // null char for empty strings to point at

  void adopt_(char* str, APT_Int32 len);

  APT_String(char* str, APT_Int32 len, int adoptFlag);
  // special ctor form that takes ownership of str
```

```
static void delfunc(void*);
static void protocolStorageManagementInfo(bool* needManaging,
                                          APT_UInt32* offsetOfFlag,
                                          int* bitNumOfFlag,
                                          bool* invertFlag,
                                          void (**delfunc)(void*));

void initFrom(const char* content, APT_Int32 len);
/*
   effect    Like assignFrom, except assumes that this string instance
             is default constructed (empty).
*/

void dtor();

void adopt_badarg();

bool isEmpty_fixed() const;
const char* data_nonOwn() const;

void badIndex(APT_Int32 index) const;

static APT_String append2(const char* str1, APT_Int32 len1,
                          const char* str2, APT_Int32 len2);

void allocBuf(APT_Int32 len, const char* src, APT_Int32 srcLen);
void allocBuf_realloc(APT_Int32 len, const char* src, APT_Int32 srcLen);

void prepareForFielding();
/* prepares this instance to be managed as a field value by the
   framework */

void clear();
// make this string empty (if var-len) or padded (if fixed-len)

union
{ char* const* basePtr_;
  char* base_;
};
union

{ char* nts_;
  APT_UInt32 allocLen_;
};

// formerly size_t to match record-size lengths, but this
// introduces padding on Alpha.  This way, it all fits in 24 bytes
// (instead of 28) even on 64-bit systems.
union
{ APT_UInt32 offset_;
  APT_UInt32 length_;
};
```

```
  // fit these within a 32-bit word
public:
    struct Bits
    {
      char padChar_ : 8;
      unsigned isFixedLength_ : 1;
      unsigned isBoundedLength_ : 1;
      unsigned notOwnBuf_ : 1;
      unsigned isBound_ : 1;
      unsigned sLength_ : 20;
    } ;
private:
  union
  { APT_UInt32 word_;
    Bits bits_;
  };

  APT_DECLARE_NEW_AND_DELETE(APT_String);
};


inline APT_String::APT_String()
 : base_(0), nts_(0), length_(0), word_(0)
{
}

inline APT_String::~APT_String()
{
  if (nts_ || base_)
    dtor();
}

inline char APT_String::padChar() const
{
  return bits_.padChar_;
}
inline void APT_String::setPadChar(char c)
{
  bits_.padChar_ = c;
}

inline APT_Int32 APT_String::length() const
{
  if (bits_.isFixedLength_) return bits_.sLength_;
  else return length_;
}

inline char* APT_String::content()
{
  if (bits_.isFixedLength_)
    return *basePtr_+offset_;
```

```
  else
    return length_ ? base_ : 0;
}
inline const char* APT_String::content() const
{
  return ((APT_String*) this)->content();
}
inline char* APT_String::nonTerminatedContent()
{
  return content();
}
inline const char* APT_String::nonTerminatedContent() const
{
  return content();
}

inline APT_String::operator const char* () const
{
  return data();
}
inline const char* APT_String::terminatedContent() const
{
  return data();
}

inline char APT_String::operator[] (APT_Int32 index) const
{
  if (index == length()) return 0;
  else return content()[index];
}

#ifdef APT_INT32_IS_NOT_INT
inline char APT_String::operator[] (int index) const
{
 return operator[]((APT_Int32) index);
}
#endif

inline char APT_String::operator[] (unsigned int index) const
{
 return operator[]((APT_Int32) index);
}

inline char APT_String::getChar(APT_Int32 index) const
{
  APT_Int32 len = length();
  if (index < 0 || index > len)
    badIndex(index);

  if (index == len) return 0;
  else return content()[index];
}
```

```
inline void APT_String::clear()
{
  if (isFixedLength())
    memset(*basePtr_+offset_, bits_.padChar_, bits_.sLength_);
  else
    length_ = 0;                // keep allocation, if any
}

inline APT_Archive& operator<< (APT_Archive& ar, const APT_String& d)
{
  return ar || (APT_String&) d;
}
inline APT_Archive& operator>> (APT_Archive& ar, APT_String& d)
{
  return ar || d;
}


class APT_StringAccum
/* Helper class for efficiently performing repeated appends onto
   a string accumulator.
*/
{
public:
  APT_StringAccum();
  ~APT_StringAccum();

  void operator+= (char c)
    { reserve(size_+1); buf_[size_++] = c; }
  void operator+= (const char*);
  void operator+= (const APT_String&);
  void operator+= (const APT_StringAccum&);

  bool operator== (const char*) const;
  bool operator!= (const char* str) const { return !(*this == str); }

  void reset()
    { size_ = 0; }

  const char* content() const
    { APT_StringAccum* ncThis = (APT_StringAccum*) this;
      ncThis->reserve(size_+1);
      ncThis->buf_[size_] = 0;
      return buf_;
    }

  int length() const { return size_; }

  operator const char*() const { return content(); }

  APT_String string() const { return APT_String(buf_, size_); }
```

```
private:
  APT_StringAccum(const APT_StringAccum&);
  APT_StringAccum& operator= (const APT_StringAccum&);

  void reserve(int neededSize)
  { if (neededSize > allocSize_) realloc(neededSize); }

  void realloc(int neededSize);

  enum { kSmallBuf=16 };
  char small_[kSmallBuf];

  int size_;
  int allocSize_;
  char* buf_;

  APT_DECLARE_NEW_AND_DELETE(APT_StringAccum);
};


#endif // APT_String_H
```

```
// -*-Mode: C++-*-
// Copyright (c) 1997 Torrent Systems, Inc.  All rights reserved.

#ifndef APT_TIME_H
#define APT_TIME_H

#ifndef APT_INTS_H
#include <apt_util/ints.h>
#endif

#ifndef APT_BOOL_H
#include <apt_util/bool.h>
#endif

#ifndef APT_DATE_H
#include <apt_util/date.h>
#endif

#ifndef APT_FAST_ALLOC_H
#include <apt_util/fast_alloc.h>
#endif

#include <sys/types.h>

class APT_Archive;

/* The APT_Time class represents the time of day with either 1 second
   resolution or microsecond resolution.  Time values range from
   00:00:00 to 23:59:59.999999; advancing time past its largest value
   wraps to its smallest value.

   No time zone information is carried by APT_Time; all time values
   are assumed to be in the same time zone, presumably GMT (UTC).  No
   provision is made for leap-seconds.

   This class can parse a limited set of string formats, all
   representing the parts of their data as numeric.  Specifically, any
   format string constructed as follows:

     %hh               Two digit hours; must be zero padded if < 10
     %nn               Two digit minutes; must be zero padded if < 10
     %ss               Two digit seconds; must be zero padded if < 10
     %ss.N             Two digit seconds plus fractional part; must
                       be zero padded if < 10
                       N is the number of fractional digits between 0
                       and 6; if 0, then no decimal point is printed.
                       Trailing zeros are not suppressed.

   The minutes format character is 'n' rather than 'm' to avoid
   confusion with APT_Date's month format character.

   All other characters match anything on input, and are output as
   themselves on output.  ISO format (?) would thus be "%hh:%nn:%ss".
```

```
   Note that all of these are fixed width formats which require zero
   padding.  A possible future enhancement is to allow on both input
   and output both space padded hours, minutes, and seconds, and no
   padding at all.  These additions might make parsing somewhat less
   efficient.

   Another future enhancement is to incorporate time zone
   specification into the format string.

   Another possible future expansion might be to have a "fast parse"
   that didn't do the error checking to make sure that it was passed a
   valid time string.

   The APT_TimeStamp class is logically the concatentation of APT_Date
   and APT_Time.  For APT_TimeStamp, the APT_Date and APT_Time string
   format strings may be combined in any manner as long as all
   APT_Date components lexically precede (or follow) all APT_Time
   components.

   These classes are designed to accommodate the corresponding RDBMS
   data types.
*/

class APT_TimeStamp;


class APT_Time
{
public:
  // define copy/assign
  APT_Time(const APT_Time&);
  ~APT_Time();

  APT_Time& operator= (const APT_Time&);
  // copies value but not hasMicro attribute

  APT_Time(bool hasMicro=false);
  /*
     effect    Creates a time object set to 00:00:00.000000 GMT.
  */

  APT_Time(int hour, int minute, APT_DFloat second, bool hasMicro=false);
  /*
     effect    Constructs a time object with the given hour, minute,
               and second, GMT.
               If the arguments are invalid, then an invalid time
               object is produced.
     validity  0 <= hour <= 23
               0 <= minute <= 59
               0 <= second <= 59.999999
     TBD: optional timezone arg?
  */
```

```
 APT_Time(int hour, int minute, int second, APT_Int32 microseconds, bool hasMicro);
  /*
     effect    Constructs a time object with the given hour, minute,
               and second, and microsecod GMT.
               If the arguments are invalid, then an invalid time
               object is produced.
     validity  0 <= hour <= 23
               0 <= minute <= 59
               0 <= second <= 59
               0 <= microseconds <= 999999
     TBD: optional timezone arg?
  */

  APT_Time(const char* timeString,
           const char* formatString="%hh:%nn:%ss", bool hasMicro=false);
  /*
     effect    Creates a time object from parsing the given string.
               If the string doesn't match the given format, an invalid
               time object is created.
     requires  formatString must be a valid format (see above).
     note      The string need not be null terminated; the
               formatString fully specifies the parsing without
               regard to null characters.
     TBD: optional timezone arg?  Or is this part of the format string?
  */

  void set(const char *timeString,
           const char *formatString = 0);
  // effect     Sets this time to the time represented in timeString
  //            A parse error will result in this time being set invalid.
  // requires   The format string must be valid.
  // note       The string need not be null terminated; the
  //            formatString fully specifies the parsing without
  //            regard to nulls.

private:
  void setHasMicrosecondResolution(bool);
public:
  bool hasMicrosecondResolution() const { return hasMicro_; }
  /*
     effect    Indicates whether APT_Time records microsecond
               resolution (true) or just second resolution (false).
               False is the default.
  */

  void setFromSeconds(APT_DFloat secondsFromMidnight);
  /*
     effect    Sets this time from the given number of seconds from
               midnight, GMT.
     note      Wrap-around logic is applied, so an argument of 86400
               (24*60*60) results in a time value of 00:00:00.
               Negative values are correctly interpreted, so an
               argument of -1 results in 23:59:59.
     TBD: optional timezone arg?
```

```
*/

void setFromHoursMinutesSecondsMicroseconds(int hour, int minute,
                                            int second, APT_Int32 microseconds);
/*
  validity  0 <= hour <= 23
            0 <= minute <= 59
            0 <= second <= 59
            0 <= microseconds <= 999999
*/

bool isValid() const;
/*
   effect    Tells whether this object represents a valid time.
             Returns false if this object was parsed from an invalid
             string, or set from an invalid set of hour, minute,
             second values.
*/

APT_String asString(const char* format=0) const;
/*
   effect    Returns a string representation of the time, formatted
             according to the passed argument.
             An invalid time will be represented by a string of '*'s
   TBD: optional timezone arg?  Or is this part of the format string?
*/

APT_DFloat secondsFromMidnight() const;
/*
   effect    Returns this time's value as the number of seconds from
             the preceding midnight, GMT.  The returned value is
             never negative.
   requires  isValid() == true
   TBD: optional timezone arg?
*/

APT_DFloat microsecondsFromMidnight() const;
/*
   effect    Returns this time's value as the number of seconds from
             the preceding midnight, GMT.  The returned value is
             never negative.
   requires  isValid() == true
   TBD: optional timezone arg?
*/

int hours() const;
int minutes() const;
int seconds() const;
APT_DFloat secondsAndFraction() const;
APT_Int32 microSeconds() const;
/*
   effect    Returns the specified portion of the time, GMT.
   requires  isValid() == true
   TBD: optional timezone arg?
```

```
*/

void addHours(int hours, int* dayCarry=0);
void addMinutes(int minutes, int* dayCarry=0);
void addSeconds(APT_DFloat seconds, int* dayCarry=0);
void addMicroSeconds(APT_Int32 microSeconds, int* dayCarry=0);
/*
   effect    Adds the indicated quantity to this time object.
             The optional dayCarry arg is incremented or decremented
             by the number of days advanced or rolled back.  For
             example, a call of addHours(24) always sets *dayCarry
             to 1; if -1 hours are added to a time value of
             00:00:00, then *dayCarry is set to -1.
   note      Carry/wraparound and negative values are handled.
             If isValid() == false, then there is no effect.
*/

/* TBD: arithmetic with SQL-style datetime intervals. */

static int compare(const APT_Time& t1, const APT_Time& t2);
/*
   effect    Returns -1, 0, or +1 according to whether t1 is less
                 than, equal to, or greater than t2.
   requires  Both times must have isValid() == true
*/

friend bool operator== (const APT_Time& op1, const APT_Time& op2);
friend bool operator!= (const APT_Time& op1, const APT_Time& op2);
friend bool operator< (const APT_Time& op1, const APT_Time& op2);
friend bool operator<= (const APT_Time& op1, const APT_Time& op2);
friend bool operator> (const APT_Time& op1, const APT_Time& op2);
friend bool operator>= (const APT_Time& op1, const APT_Time& op2);
// note: The secondsFromMidnight() value is the basis of the comparison
// requires: isValid() true for both arguments

APT_UInt32 hash() const;
/*
   effect    Returns a value well suited for hashing.
   requires  isValid() == true
*/

static APT_Time now(bool hasMicro=false);
/*
   effect    Returns a time representing the time right now, GMT.
   TBD: optional timezone arg?
*/

static bool isFormatValid(const char* format);
/*
   effect    Returns true if format points to a valid format, false
             otherwise.  The valid formats are specified above.
*/

/* Support for compiled parsing and printing. */
```

```
class ParseObject
{
public:
  ParseObject(const char* format);
  /*
     effect    Creates a parse object that describes how to parse
               the given format in a "compiled" form; this object
               may be used to parse multiple instances of the given
               format.
     requires  format must be a valid format string (see above).
  */

  ParseObject();
  /*
     effect    Creates a parse object which parses "hh:nn:ss" times
               (our default).
     note      This is separate from the above to avoid the
               need to parse time formats in the default case.
  */

  // persistence
  friend APT_Archive& operator ||(APT_Archive& ar, ParseObject& o);

  APT_Time parse(const char* timeString) const
  { return parse_(timeString); }
  void parse(const char* timeString, APT_Time* result) const
  { parse_(timeString, result); }
  /*
     effect    Parses the time according to the format given to the
               constructor.  Returns a time with isValid() == false
               if the parse fails.
               A possible future optimization might be to provide a
               fast parse routine that didn't do the error checking
               implied in this guarantee.
     requires  timeString must be at least as long as the format
               string specifies.
  */

  void parseAndSet(const char* &timeString, APT_Time &time) const;

  /*  effect   Parses the time according to the format given to the
               constructor.  Sets time with isValid() == false
               if the parse fails.  Otherwise, sets the time
               parsed from timeString.
      requires  timeString must be at least as long as the format
               string specifies.
  */

  void set(const char* format);
  /*
     effect    Sets the parse object to describe the given format.
     requires  Format must be a valid format string.
  */
```

```
   APT_String format() const;
   // effect   Return the format corresponding to the parseObject

   APT_String toString(const APT_Time& time) const;
   /*
      effect    Returns a string representing the time specified in the
                format given to the constructor of this object.  An
                invalid time will be represented by a string of '*'s
   */

   /*
    * String creaters optimized for efficiency.
    */

   APT_UInt32 printSize(const APT_Time& time) const;
   /*
      effect    Returns the buffer size the specified time will
                require.
   */

   APT_UInt32 maxPrintSize() const;
   /*
      effect    Returns the maximum size that any time may take using
                the format represented by this object.
   */

   bool isPrintFixedSize() const;
   /*
      effect    Returns true if the format represented by this object
                will always result in a fixed sized print
                representation (i.e. the result of printSize() will be
                independent of the argument, and that result will be
                equal to maxPrintSize()).
   */

   void printToBuf(const APT_Time& time, char* dest, int* length) const;
   /*
      effect    Writes a printable representation of the time into
                the buffer according to the format represented
                by this object.  If length is non-null, the
                amount of space used will be written to the
                location to which it points.  No null will be
                written at the end of the representation.
                If an invalid time is passed, the buffer will be
                filled with '*'s.
      requires  dest must point to a buffer that is at least
                printSize() long.
   */

   bool isThisFormatValid() const;
   /*
      effect    Returns true if this parseobject represents a valid format,
                false otherwise.
```

```
    */

    static bool isFormatValid(const char* format);
    /*
       effect    Returns true if format points to a valid format, false
                 otherwise.
    */

    // Default destructor, copy constructor, and assignment operator
    // ok.

  private:
    friend class APT_TimeStamp;

    static bool parseFormat(const char* format, ParseObject* results=0);
    /*
       effect    Parses the specified format returning true if it is a
                 valid format and false if not.  If the optional
                 results argument is non-null, it is filled in with the
                 results of parsing the format.
    */

    APT_Time parse_(const char* &timeString) const;
    void parse_(const char* &timeString, APT_Time* result) const;
    // The timeString pointer is incremented past the consumed input.
public:
    int hourOffset_;
    int minutesOffset_;
    int secondsOffset_;
    int secondsFractional_;
    APT_String startingString_;
  };

  // persistence
  friend APT_Archive& operator ||(APT_Archive& ar, APT_Time& d);

private:
  friend class APT_Time::ParseObject;
  friend class APT_TimeStamp;
  friend class APT_TimeDescriptor; /* for access to private function to
                                          set hasMicro */

  // rep is for Orchestrate base/offset protocol
  /* 3 bytes or 6 bytes (with microsecond resolution) */

public:
  struct TimeRep
  {
    APT_UInt8 hour_;
    APT_UInt8 minute_;
    APT_UInt8 second_;
    APT_UInt8 microHi_;         // present iff hasMicro_ true
    APT_UInt8 microMid_;        // ditto
    APT_UInt8 microLo_;         // ditto
```

```
  };


  const TimeRep* rep() const { return (const TimeRep*) (*basePtr_ + offset_); }
  TimeRep* rep() { return (TimeRep*) (*basePtr_ + offset_); }

  APT_Time(const TimeRep** timeRepPtrPtr,
           bool hasMicro)
    : hasMicro_(hasMicro) {

      ourAlloc_ = 0;
      basePtr_ = (char* const *) timeRepPtrPtr;
      offset_ = 0;
    }
  /*
     effect    Creates a temporary time object from a TimeRep structure and a
hasMicro
               flag.
     requires  resulting APT_Time object only be used temporarily within
               the scope of the constructing function.  For temporary use
               only, as in APT_TimeStamp.
     WARNING   TO BE USED BY TORRENT ONLY.
  */
private:
  //APT_Time(const TimeRep*, bool hasMicro);
  /*
     effect    Creates a time object from a TimeRep structure and a boolean.
               The TimeRep is not copied and as such objects created by this
               constructor should be used as temporary objects only.
   */



  void releaseStorage();

  bool hasMicro_;

  char* const* basePtr_;
  APT_UInt32 offset_;
  char* ourAlloc_;

  static ParseObject spoDefaultNoFraction_;
  static ParseObject spoDefaultFraction_;

  APT_DECLARE_NEW_AND_DELETE(APT_Time);
};
APT_DIRECTIONAL_SERIALIZATION(APT_Time);
APT_DIRECTIONAL_SERIALIZATION(APT_Time::ParseObject);


class APT_TimeStamp
{
public:
  APT_TimeStamp(bool hasMicro=false);
```

```
/*
   effect    Creates a timestamp for which isValid() is false.
             A valid timestamp value must be assigned to this object
             before its value can be used.
*/

// Copy ctor and assignment must be provided because base, offset protocol

APT_TimeStamp(const APT_TimeStamp&);
~APT_TimeStamp();

APT_TimeStamp& operator= (const APT_TimeStamp&);
// copies value but not hasMicro attribute


APT_TimeStamp(const APT_Date& date,
              int hour, int minute, APT_DFloat second, bool hasMicro=false);
/*
   effect    Constructs a timestamp object with the given date,
             hour, minute, and second, GMT.
             If the arguments are invalid, then an invalid timestamp
             object is produced.
   validity  date.isValid()
             0 <= hour <= 23
             0 <= minute <= 59
             0 <= second <= 59.999999
   TBD: optional timezone arg?
*/

APT_TimeStamp(const APT_Date&, const APT_Time&, bool hasMicro=false);
/*
  effect    Constructs a timestamp object with the given date and
            time, GMT.
            If the arguments are invalid, then an invalid timestamp
            object is produced.
*/

APT_TimeStamp(const char* timeStampString,
              const char* formatString="%yyyy-%mm-%dd %hh:%nn:%ss",
              bool hasMicro=false);
/*
   effect    Creates a timestamp object from parsing the given string.
             If the string doesn't match the given format, an invalid
             timestamp object is created.
   requires  formatString must be a valid format.
   note      The string need not be null terminated; the
             formatString fully specifies the parsing without
             regard to null characters.
   TBD: optional timezone arg?  Or is this part of the format string?
*/

/* the signature here wants to be in terms of 'time_t', but because
   we build on one version of AIX (4.1), and ship to later versions,
   this is a problem because IBM canged the typedef for time_t from
```

```
   long to int. */
void setFromTimeT(APT_Int32);
APT_Int32 asTimeT() const;
/*
   effect    Converts this timestamp from/to a time_t as returned by
             the Unix time(2) subroutine.
*/

bool isValid() const;
/*
   effect    Tells whether this object represents a valid timestamp.
             Returns false if this object was parsed from an invalid
             string, or set from an invalid date or time object.
*/

bool hasMicrosecondResolution() const { return hasMicro_; }
/*
   effect    Indicates whether APT_TimeStamp records microsecond
   resolution (true) or just second resolution (false).
   False is the default.
*/

APT_String asString(const char*
                          format=0) const;
/*
   effect    Returns a string representation of the timestamp, formatted
             according to the passed argument.  If format=0, the default,
             then one of two default formats will be used, depending on
             whether or not the timestamp has microsecond resolution.
             An invalid timestamp will be represented by a string of '*'s
   TBD: optional timezone arg?  Or is this part of the format string?
*/

void addHours(int hours);
void addMinutes(int minutes);
void addSeconds(APT_DFloat seconds);
void addMicroSeconds(APT_Int32 microSeconds);
/*
   effect    Adds the indicated quantity to this timestamp object.
   note      Carry/wraparound and negative values are handled.
             If isValid() == false, then there is no effect.
*/

APT_Date date() const { return rep()->date_; }
APT_Time time() const;
void getDate(APT_Date*) const;
void getTime(APT_Time*) const;
/*
   effect    Returns the specified portion of the timestamp.
             If isValid() is false, then the returned object will be
             invalid as well.
   note      The latter two forms are more efficient.
   TBD: optional timezone arg?
*/
```

```
void setDate(const APT_Date& rhs) { rep()->date_ = rhs; }
void setTime(const APT_Time&);
/*
   effect    Sets the corresponding portion of this timestamp.
             If the argument is invalid, isValid() becomes false for
             this timestamp.
   TBD: optional timezone arg?
*/

void setFromYMDHMSAndMicroseconds(int years, int months, int days,
                                  int hours, int minutes, int seconds,
                                  APT_Int32 microseconds);

/*
   effect    Sets the entire timestamp.  if the argument is invalid,
             isValid() becomes false for the timestamp.
*/

APT_DFloat secondsFromMidnight() const;
/*
   effect    Returns this timestamp's value as the number of seconds from
             the preceding midnight, GMT.  The returned value is
             never negative.
   requires  isValid() == true
   TBD: optional timezone arg?
*/
friend bool operator== (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
friend bool operator!= (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
friend bool operator< (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
friend bool operator<= (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
friend bool operator> (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
friend bool operator>= (const APT_TimeStamp& op1, const APT_TimeStamp& op2);
// requires: isValid() true for both arguments

APT_UInt32 hash() const;
/*
   effect    Returns a value well suited for hashing.
   requires  isValid() == true
*/

static int compare(const APT_TimeStamp& ts1, const APT_TimeStamp& ts2);
/*
   effect    Returns -1, 0, or +1 according to whether ts1 is less
             than, equal to, or greater than ts2.
   requires  Both timestamps must have isValid() == true
*/

static APT_TimeStamp now(bool hasMicro=false);
/*
   effect    Returns a timestamp representing the time right now.
*/

static bool isFormatValid(const char* format);
```

```
/*
   effect    Returns true if format points to a valid format, false
             otherwise.
*/

void set(const char *timeStampString,
         const char *formatString = 0);
// effect     Sets this timestamp to the timestamp represented in timeStampString
//            A parse error will result in this time being set invalid.
// requires   The format string must be valid or 0 in which case one of two
//            default strings will be used depending on hasMicro_.
//            If hasMicro_ is true a format string showing a fractional component
//            is selected.  Otherwise not.
// note       The string need not be null terminated; the
//            formatString fully specifies the parsing without
//            regard to nulls.

/* Support for compiled parsing and printing. */

class ParseObject
{
public:
  enum StreamType {eformatError= -2, eUnknown=-1, eDate=0, eTime=1};

  ParseObject(const char* format);
  /*
     effect    Creates a parse object that describes how to parse
               the given format in a "compiled" form; this object
               may be used to parse multiple instances of the given
               format.
     requires  format must be a valid format string (see above).
  */

  ParseObject();
  /*
     effect    Creates a parse object which parses "yyyy-mm-dd hh:nn:ss"
               timestamps (our default).
     note      This is separate from the above to avoid the
               need to parse time formats in the default case.
  */

  // persistence
  friend APT_Archive& operator ||(APT_Archive& ar, ParseObject& o);

  APT_TimeStamp parse(const char* timeStampString) const;
  /*
     effect    Parses the time according to the format given to the
               constructor.  Returns a timestamp with isValid() == false
               if the parse fails.
               A possible future optimization might be to provide a
               fast parse routine that didn't do the error checking
               implied in this guarantee.
     requires  timeStampString must be at least as long as the format
               string specifies.
```

```
   */

   void parse(const char* timeStampString, APT_TimeStamp* timeStampPtr);
   /*
      effect    Parses the time according to the format given to the
                constructor.  Modifies the timestamp pointed to by
                timeStampPtr with isValid() == false
                if the parse fails.
                A possible future optimization might be to provide a
                fast parse routine that didn't do the error checking
                implied in this guarantee.
      requires  timeStampString must be at least as long as the format
                string specifies.
   */

   void set(const char* format);
   /*
      effect    Sets the parse object to describe the given format.
      requires  Format must be a valid format string.
   */

   APT_String format() const;
   // effect   Return the format corresponding to the parseObject

   APT_String toString(const APT_TimeStamp& timeStamp) const;
   /*
      effect    Returns a string representing the timestamp specified
                in the format given to the constructor of this
                object.  An invalid timestamp will be represented by
                a string of '*'s
   */

   /*
    * String creaters optimized for efficiency.
    */

   APT_UInt32 printSize(const APT_TimeStamp& timeStamp) const;
   /*
      effect    Returns the buffer size the specified timestamp will
                require.
   */

   APT_UInt32 maxPrintSize() const;
   /*
      effect    Returns the maximum size that any timestamp may take using
                the format represented by this object.
   */

   bool isPrintFixedSize() const;
   /*
      effect    Returns true if the format represented by this object
                will always result in a fixed sized print
                representation (i.e. the result of printSize() will be
                independent of the argument, and that result will be
```

```
                equal to maxPrintSize()).
    */

    void printToBuf(const APT_TimeStamp& timeStamp, char* dest,
                    int* length) const;
    /*
       effect    Writes a printable representation of the timestamp into
                 the buffer according to the format represented
                 by this object.  If length is non-null, the
                 amount of space used will be written to the
                 location to which it points.  No null will be
                 written at the end of the representation.
                 If an invalid timestamp is passed, the buffer will be
                 filled with '*'s.
                 The dest pointer is incremented past the produced
                 output.
       requires  dest must point to a buffer that is at least
                 printSize() long.
    */

    static bool isFormatValid(const char* format);
    /*
       effect    Returns true if format points to a valid format, false
                 otherwise.
    */

    // Default destructor, copy constructor, and assignment operator
    // ok.

  private:
    static bool parseFormat(const char* format, ParseObject* results=0);
    /*
       effect    Parses the specified format returning true if it is a
                 valid format and false if not.  If the optional
                 results argument is non-null, it is filled in with the
                 results of parsing the format.
    */

    APT_Date::ParseObject dateParse_;
    APT_Time::ParseObject timeParse_;
    int firstParseType_;
  };

  // persistence
  friend APT_Archive& operator ||(APT_Archive& ar, APT_TimeStamp& d);

private:
  friend class APT_TimeStamp::ParseObject;
  friend class APT_TimeStampDescriptor; /*for access to private function to set
hasMicro*/
  friend class APT_GFIX_TimeStamp; /*for binary import export of stamp rep*/

  void setHasMicrosecondResolution(bool microsecResFlag);
```

```
  APT_Int32 intSecondsFromMidnight() const;
  /*
     effect    Returns this time's value as the number of seconds from
               the preceding midnight, GMT.  The returned value is
               never negative.
     requires  isValid() == true
     TBD: optional timezone arg?
  */

  void setFromJulianDaysAndSecondsFromMidnight(APT_Int32 julianDays, APT_Int32
secondsFromMidnight);
  /*
     effect    sets the timestamp using julian days and seconds from midnight, GMT.
  */

  // rep is for Orchestrate base/offset protocol
  /* 7 bytes or 10 bytes (with microsecond resolution) */

  struct StampRep
  {
    APT_Date date_;             // aligned x4
    APT_Time::TimeRep time_;    // 3 or 6 bytes, depending on hasMicro_
  };

  const StampRep* rep() const
    { return (const StampRep*) (*basePtr_ + offset_); }
  StampRep* rep()
    { return (StampRep*) (*basePtr_ + offset_); }

  void releaseStorage();

  bool hasMicro_;

  char* const* basePtr_;
  APT_UInt32 offset_;

  char* ourAlloc_;
  static ParseObject spoDefaultNoFraction_;
  static ParseObject spoDefaultFraction_;

  APT_DECLARE_NEW_AND_DELETE(APT_TimeStamp);
};

// Support for output to stream.
ostream& operator<< (ostream&, const APT_Time&);
ostream& operator<< (ostream&, const APT_TimeStamp&);

// Persistence
APT_DIRECTIONAL_SERIALIZATION(APT_TimeStamp);
APT_DIRECTIONAL_SERIALIZATION(APT_TimeStamp::ParseObject);

#endif // APT_TIME_H
```