Chapter 5

C++ Binding

5.1 Introduction

This chapter defines the C++ binding for ODL/OML.

ODL stands for Object Definition Language. It is the declarative portion of C++ ODL/OML. The C++ binding of ODL is expressed as a library that provides classes and functions to implement the concepts defined in the ODMG Object Model. OML stands for Object Manipulation Language. It is the language used for retrieving objects from the database and modifying them. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library.

ODL/OML specifies only the logical characteristics of objects and the operations used to manipulate them. It does not discuss the physical storage of objects. It does not address the clustering or memory management issues associated with the stored physical representation of objects or access structures like indices used to accelerate object retrieval. In an ideal world these would be transparent to the programmer. In the real world they are not. An additional set of constructs called *physical pragmas* is defined to give the programmer some direct control over these issues, or at least to enable a programmer to provide "hints" to the storage management subsystem provided as part of the ODBMS runtime. Physical pragmas exist within the ODL and OML. They are added to object type definitions specified in ODL, expressed as OML operations, or shown as optional arguments to operations defined within OML. Because these pragmas are not in any sense a stand-alone language, but rather a set of constructs added to ODL/OML to address implementation issues, they are included within the relevant subsections of this chapter.

The chapter is organized as follows. Section 5.2 discusses the ODL. Section 5.3 discusses the OML. Section 5.4 discusses OQL—the distinguished subset of OML that supports associative retrieval. Associative retrieval is access based on the values of the properties of objects rather than on their IDs or names. Section 5.6 provides an example program.

5.1.1 Language Design Principles

The programming language–specific bindings for ODL/OML are based on one basic principle: The programmer feels that there is one language, not two separate languages with arbitrary boundaries between them. This principle has two corollaries that are evident in the design of the C++ binding defined in the body of this chapter:

1. There is a single unified type system across the programming language and the database; individual instances of these common types can be persistent or transient.
2. The programming language–specific binding for ODL/OML respects the syntax and semantics of the base programming language into which it is being inserted.

5.1.2 Language Binding

The C++ binding maps the Object Model into C++ by introducing a set of classes that can have both persistent and transient instances. These classes are informally referred to as "persistence-capable classes" in the body of this chapter. These classes are distinct from the normal classes defined by the C++ language, all of whose instances are transient; that is, they don't outlive the execution of the process in which they were created. Where it is necessary to distinguish between these two categories of classes, the former are called "persistence-capable classes"; the latter are referred to as "transient classes."

The C++ to ODBMS language binding approach described by this standard is based on the smart pointer or "Ref-based" approach. For each persistence-capable class T, an ancillary class d_Ref<T> is defined. Instances of persistence-capable classes are then referenced using parameterized references, e.g.,

```
(1) d_Ref<Professor>     profP;
(2) d_Ref<Department>    deptRef;
(3) profP–>grant_tenure();
(4) deptRef = profP–>dept;
```

Statement (1) declares the object profP as an instance of the type d_Ref<Professor>. Statement (2) declares deptRef as an instance of the type d_Ref<Department>. Statement (3) invokes the grant_tenure operation defined on class Professor, on the instance of that class referred to by profP. Statement (4) assigns the value of the dept attribute of the professor referenced by profP to the variable deptRef.

Instances of persistence-capable classes may contain embedded members of C++ built-in types, user-defined classes, or pointers to transient data. Applications may refer to such embedded members using C++ pointers (*) or references (&) only during the execution of a transaction.

In this chapter we use the following terms to describe the places where the standard is formally considered undefined or allows for an implementor of one of the bindings to make implementation-specific decisions with respect to implementing the standard. The terms are

> *Undefined:* The behavior is unspecified by the standard. Implementations have complete freedom (can do anything or nothing), and the behavior need not be documented by the implementor or vendor.
>
> *Implementation-defined:* The behavior is specified by each implementor/vendor. The implementor/vendor is allowed to make implementation-specific decisions about the behavior. However, the behavior must be well defined and fully documented and published as part of the vendor's implementation of the standard.

Figure 5-1 shows the hierarchy of languages involved, as well as the preprocess, compile, and link steps that generate an executable application.

5.1.3 Mapping the ODMG Object Model into C++

Although C++ provides a powerful data model that is close to the one presented in Chapter 2, it is worth trying to explain more precisely how concepts introduced in Chapter 2 map into concrete C++ constructs.

5.1.3.1 Object and Literal

An ODMG object type maps into a C++ class. Depending on how a C++ class is instantiated, the result can be an ODMG object or an ODMG literal. A C++ object embedded as a member within an enclosing class is treated as an ODMG literal. This is explained by the fact that a block of memory is inserted into the enclosing object and belongs entirely to it. For instance, one cannot copy the enclosing object without getting a copy of the embedded one at the same time. In this sense the embedded object cannot be considered as having an identity, since it acts as a literal.

5.1.3.2 Structure

The Object Model notion of a *structure* maps into the C++ construct *struct* or *class* embedded in a class.

5.1.3.3 Implementation

C++ has implicit the notion of dividing a class definition into two parts: its interface (public part) and its implementation (protected and private members and function definitions). However, in C++ only one implementation is possible for a given class.

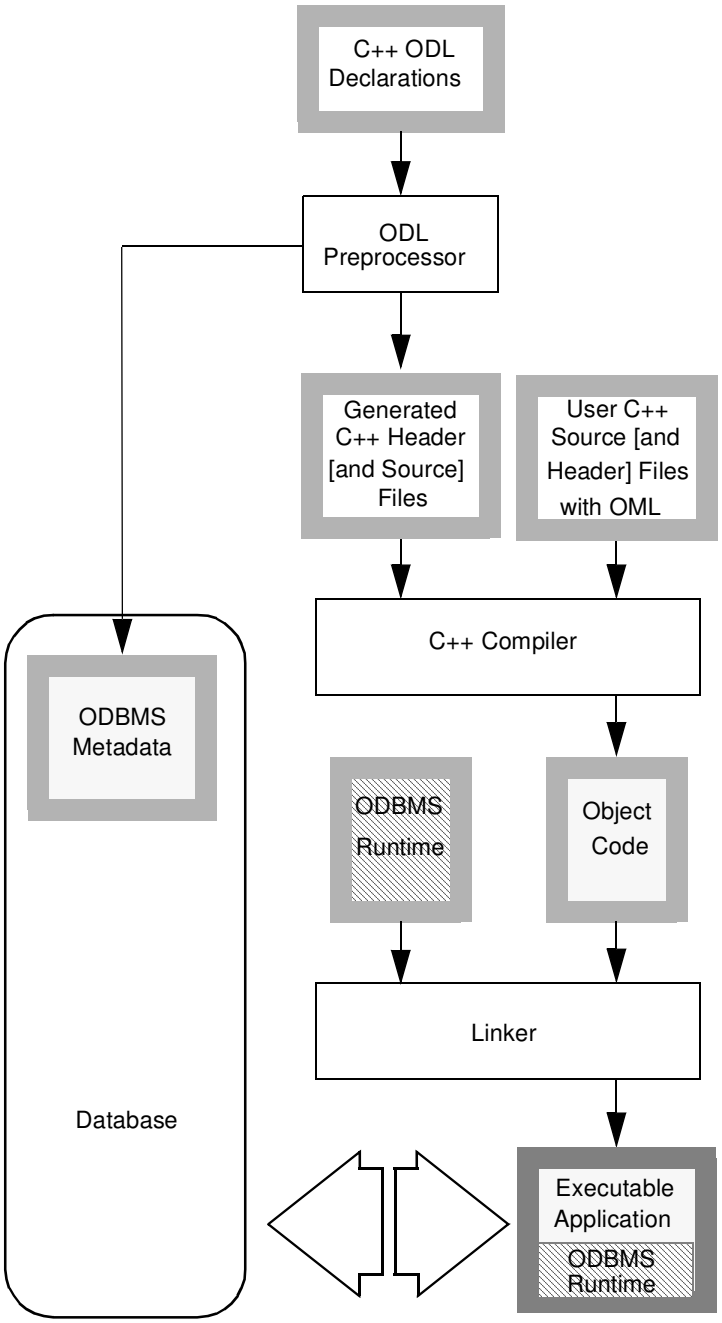


Figure 5-1. Language Hierarchy

5.1.3.4 Collection Classes

The ODMG Object Model includes collection type generators, collection types, and collection instances. Collection type generators are represented as *template classes* in C++. Collection types are represented as collection classes, and collection instances are represented as instances of these collection classes. To illustrate these three categories:

```
template<class T> class d_Set : public d_Collection<T> { ... };
class Ship { ... };
d_Set<d_Ref<Ship> > Cunard_Line;
```

d_Set<T> is a collection template class. d_Set<d_Ref<Ship> > is a collection class. And Cunard_Line is a particular collection, an instance of the class d_Set<d_Ref<Ship> >.

The subtype/supertype hierarchy of collection types defined in the ODMG Object Model is directly carried over into C++. The type d_Collection<T> is an abstract class in C++ with no direct instances. It is instantiable only through its derived classes. The only differences between the collection classes in the C++ binding and their counterparts in the Object Model are the following:

- Named operations in the Object Model are mapped to C++ function members.
- For some operations, the C++ binding includes both the named function and an overloaded infix operation, e.g., d_Set::union_with also has the form operator+=. The statements s1.union_with(s2) and s1 += s2 are functionally equivalent.
- Operations that return a boolean in the Object Model are modeled as function members that return a d_Boolean in the C++ binding.
- The create and delete operations defined in the Object Model have been replaced with C++ constructors and destructors.

5.1.3.5 Array

C++ provides a syntax for creating and accessing a contiguous and indexable sequence of objects. This has been chosen to map partially to the ODMG Array collection. To complement it, a d_Varray C++ class is also provided, which implements an array whose upper bound may vary.

5.1.3.6 Relationship

Relationships are not directly supported by C++. Instead, they are supported in ODMG by including instances of specific template classes that provide the maintenance of the relationship.

The relation itself is implemented as a reference (one-to-one relation) or as a collection (one-to-many relation) embedded in the object.

5.1.3.7 Extents

The class d_Extent<T> provides an interface to the extent for a persistence-capable class T in the C++ binding.

5.1.3.8 Keys

Key declarations are not supported by C++.

5.1.3.9 Names

The previous releases only allowed one name per object in the C++ binding. Now an object can have multiple names. The bind operation in the Object Model is implemented in C++ with the set_object_name and rename_object methods to maintain backward compatibility with previous releases of the C++ binding.

5.1.4 Use of C++ Language Features

5.1.4.1 Prefix

The global names in the ODMG interface will have a prefix of d_. The intention is to avoid name collisions with other names in the global namespace. The ODMG will keep the prefix even after C++ namespaces are generally available.

5.1.4.2 Namespaces

The namespace feature added to C++ did not have generally available implementations at the time this specification was written. In the future the ODMG plans to utilize namespaces and intends to use odmg as its namespace name.

5.1.4.3 Exception Handling

When error conditions are detected, an instance of class d_Error is thrown using the standard C++ exception mechanism. Class d_Error is derived from the class exception defined in the C++ standard.

5.1.4.4 Preprocessor Identifier

A preprocessor identifier, __ODMG_93__, is defined for conditional compilation. With ODMG 2.0 the following symbol:

```
#define __ODMG__ 20
```

is defined. The value of this symbol indicates the specific ODMG release, for example, 20 (release 2.0), 21 (release 2.1), or 30 (release 3.0).

5.1.4.5 Implementation Extensions

Implementations must provide the full function signatures for all the interface methods specified in the chapter and may provide variants on these methods, with additional

parameters. Each additional parameter must have a default value. This allows applications that do not use the additional parameters to be portable.

5.2 C++ ODL

This section defines the C++ Object Definition Language. C++ ODL provides a description of the database schema as a set of object classes—including their attributes, relationships, and operations—in a syntactic style that is consistent with that of the declarative portion of a C++ program. Instances of these classes can be manipulated through the C++ OML.

Following is an example declaring type Professor:

```
extern const char _professors[ ];
extern const char _advisor [ ];

class Professor : public d_Object {
public:
// properties:
    d_UShort                                age;
    d_UShort                                id_number;
    d_String                                office_number;
    d_String                                name;
    d_Rel_Ref<Department, _professors>      dept;
    d_Rel_Set<Student, _advisor>            advisees;
// operations:
    void                                    grant_tenure();
    void                                    assign_course(Course &);
private:
            ...
};

const char _professors [ ] = "professors";
const char _advisor [ ] = "advisor";
```

The syntax for a C++ ODL class declaration is identical to a C++ class declaration. Attribute declarations map to a restricted set of C++ data member declarations. The variables _professors and _advisor are used for establishing an association between the two ends of a relationship.

Static data members of classes are not contained within each individual instance but are of static storage class. Thus static data members are not stored in the database, but are supported for persistence-capable classes. Supertypes are specified using the stan-

dard C++ syntax within the class header, e.g., class Professor : public Employee. Though this specification may use public members for ease and brevity, private and protected members are supported.

5.2.1 Attribute Declarations

Attribute declarations are syntactically identical to data member declarations within C++. Because notions of attributes as objects are not yet defined and included in this standard, attributes and data members are not and cannot be syntactically distinguished. In this standard, an attribute cannot have properties (e.g., unit of measure) and there is no way to specialize the get_value and set_value operations defined on the type (e.g., to raise an event when a value is changed).

Standard C++ syntax and semantics for class definitions are supported. However, compliant implementations need not support the following data types within persistent classes:

- unions
- bit fields
- references (&)

as members. Unions and bit fields pose problems when supporting heterogeneous environments. The semantics of references is that they are initialized once at creation; all subsequent operations are directed to the referenced object. References within persistent objects cannot be reinitialized when brought from the database into memory and their initialization value would, in general, not be valid across process boundaries. A set of special classes is defined within the ODMG specification to contain references to persistent objects.

In addition to all primitive data types, except those noted above, structures and class objects can be members. There are several structured literal types that are provided. These include:

- d_String
- d_Interval
- d_Date
- d_Time
- d_Timestamp

Examples:

```
struct University_Address {
    d_UShort    PO_box;
    d_String    university;
    d_String    city;
    d_String    state;
    d_String    zip_code;
};
```

```
class Student : public d_Object {
public:
    d_String              name;
    d_Date                birth_date;
    Phone_Number          dorm_phone;
    University_Address    address;
    d_List<d_String>      favorite_friends;
};
```

The attribute name takes a d_String as its value. The attribute dorm_phone takes a user-defined type Phone_Number as its value. The attribute address takes a structure. The attribute favorite_friends takes a d_List of d_String as its value. The following sections contain descriptions of the provided literal types.

5.2.1.1 Fixed-Length Types

In addition to the C++ built-in data types, such as the signed, unsigned, and floating point numeric data types, the following fixed-length types will be supported for use in defining attributes of persistence capable classes.

Type Name	Range	Description
d_Short	16 bit	signed integer
d_Long	32 bit	signed integer
d_UShort	16 bit	unsigned integer
d_ULong	32 bit	unsigned integer
d_Float	32 bit	IEEE Std 754-1985 single-precision floating point
d_Double	64 bit	IEEE Std 754-1985 double-precision floating point
d_Char	8 bit	ASCII
d_Octet	8 bit	no interpretation
d_Boolean	d_True or d_False	defines d_True (nonzero value) and d_False (zero value)

Unlike the C++ built-in types, these types have the same range and interpretation on all platforms and environments. Use of these types is recommended when developing applications targeted for heterogeneous environments. Note that like all other global names described in this chapter, these types will be defined within the ODMG namespace when that feature becomes available.

Any ODMG implementation that allows access to a database from applications that have been constructed with different assumptions about the range or interpretation of

the C++ built-in types may require the use of the fixed-length data types listed above when defining attributes of persistent objects. The behavior of the database system in such a heterogeneous environment when the C++ built-in types are used for persistent data attributes is undefined.

ODMG implementations will allow but not require the use of the fixed-length data types when used in homogeneous environments.

For any given C++ language environment or platform, these fixed-length data types may be defined as identical to a built-in C++ data type that conforms to the range and interpretation requirements. Since a given C++ built-in data type may meet the requirements in some environments but not in others, portable application code should not assume any correspondence or lack of correspondence between the fixed-length data types and similar C++ built-in data types. In particular, function overloads should not be disambiguated solely on the difference between a fixed-length data type and a closely corresponding C++ built-in data type. Also, different implementations of a virtual function should use signatures that correspond exactly to the declaration in the base class with respect to use of fixed-length data types versus C++ built-in data types.

5.2.1.2 d_String

The following class defines a literal type to be used for string attributes. It is intended that this class is used strictly for storing strings in the database, as opposed to being a general string class with all the functionality of a string class normally used for transient strings in an application.

Initialization, assignment, copying, and conversion to and from C++ character strings are supported. The comparison operators are defined on d_String to compare with either another d_String or a C++ character string. One can also access an element in the d_String via an index and also determine the length of the d_String.

Definition:

```
class d_String {
public:
                        d_String();
                        d_String(const d_String &);
                        d_String(const char *);
                        ~d_String();
    d_String &          operator=(const d_String &);
    d_String &          operator=(const char *);
                        operator const char *() const;
    char &              operator[](unsigned long index);
    unsigned long       length() const;
```

```
friend  d_Boolean       operator==(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator==(const d_String &sL, const char *pR);
friend  d_Boolean       operator==(const char *pL, const d_String &sR);
friend  d_Boolean       operator!= (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator!= (const d_String &sL, const char *pR);
friend  d_Boolean       operator!= (const char *pL, const d_String &sR);
friend  d_Boolean       operator<   (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator<   (const d_String &sL, const char *pR);
friend  d_Boolean       operator<   (const char *pL, const d_String &sR);
friend  d_Boolean       operator<=(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator<=(const d_String &sL, const char *pR);
friend  d_Boolean       operator<=(const char *pL, const d_String &sR);
friend  d_Boolean       operator>   (const d_String &sL, const d_String &sR);
friend  d_Boolean       operator>   (const d_String &sL, const char *pR);
friend  d_Boolean       operator>   (const char *pL, const d_String &sR);
friend  d_Boolean       operator>=(const d_String &sL, const d_String &sR);
friend  d_Boolean       operator>=(const d_String &sL, const char *pR);
friend  d_Boolean       operator>=(const char *pL, const d_String &sR);
};
```

Class d_String is responsible for freeing the string that gets returned by operator const char *.

5.2.1.3 d_Interval

The d_Interval class is used to represent a duration of time. It is also used to perform arithmetic operations on the d_Date, d_Time, and d_Timestamp classes. This class corresponds to the day-time interval as defined in the SQL standard.

Initialization, assignment, arithmetic, and comparison functions are defined on the class, as well as member functions to access the time components of its current value.

The d_Interval class accepts nonnormalized input, but normalizes the time components when accessed. For example, the constructor would accept 28 hours as input, but then calling the day function would return a value of 1 and the hour function would return a value of 4. Arithmetic would work in a similar manner.

Definition:

```
class d_Interval {
public:
                        d_Interval(int day = 0, int hour = 0,int min = 0, float sec = 0.0);
                        d_Interval(const d_Interval &);
    d_Interval &        operator=(const d_Interval &);
    int                 day() const;
```

```
    int             hour() const;
    int             minute() const;
    float           second() const;
    d_Boolean       is_zero() const;
    d_Interval &    operator+=(const d_Interval &);
    d_Interval &    operator-=(const d_Interval &);
    d_Interval &    operator*=(int);
    d_Interval &    operator/=(int);
    d_Interval      operator-() const;
friend  d_Interval  operator+(const d_Interval &L, const d_Interval &R);
friend  d_Interval  operator-(const d_Interval &L, const d_Interval &R);
friend  d_Interval  operator*(const d_Interval &L, int R);
friend  d_Interval  operator*(int L, const d_Interval &R);
friend  d_Interval  operator/ (const d_Interval &L, int R);
friend  d_Boolean   operator==(const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator!= (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator<  (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator<=(const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator>  (const d_Interval &L, const d_Interval &R);
friend  d_Boolean   operator>=(const d_Interval &L, const d_Interval &R);
};
```

5.2.1.4 d_Date

The d_Date class stores a representation of a date consisting of a year, month, and day. It also provides enumerations to denote weekdays and months.

Initialization, assignment, arithmetic, and comparison functions are provided. Implementations may have additional functions available to support converting to and from the type used by the operating system to represent a date. Functions are provided to access the components of a date. There are also functions to determine the number of days in a month, etc. The static function current returns the current date. The next and previous functions advance the date to the next specified weekday.

Definition:

```
class d_Date {
public:
    enum Weekday {
        Sunday = 0,     Monday = 1,     Tuesday = 2,     Wednesday = 3,
        Thursday = 4,   Friday = 5,     Saturday = 6
    };
    enum Month {
```

```
        January = 1,   February = 2,   March = 3,   April = 4,   May = 5,   June = 6,
        July = 7,   August = 8,   September = 9,   October = 10,   November = 11,
        December = 12
    };
                        d_Date();           // sets to current date
                        d_Date(unsigned short year, unsigned short day_of_year);
                        d_Date(unsigned short year, unsigned short month,
                                unsigned short day);
                        d_Date(const d_Date &);
                        d_Date(const d_Timestamp &);
    d_Date &            operator=(const d_Date &);
    d_Date &            operator=(const d_Timestamp &);
    unsigned short      year() const;
    unsigned short      month() const;
    unsigned short      day() const;
    unsigned short      day_of_year() const;
    Weekday             day_of_week() const;
    Month               month_of_year() const;
    d_Boolean           is_leap_year() const;
static d_Boolean        is_leap_year(unsigned short year);
static d_Date           current();
    d_Date &            next(Weekday);
    d_Date &            previous(Weekday);
    d_Date &            operator+=(const d_Interval &);
    d_Date &            operator+=(int ndays);
    d_Date &            operator++();           // prefix  ++d
    d_Date              operator++(int);        // postfix  d++
    d_Date &            operator-=(const d_Interval &);
    d_Date &            operator-=(int ndays);
    d_Date &            operator--();           // prefix --d
    d_Date              operator--(int);        // postfix  d--
friend d_Date           operator+(const d_Date &L, const d_Interval &R);
friend d_Date           operator+(const d_Interval &L, const d_Date &R);
friend d_Interval       operator-(const d_Date &L, const d_Date &R);
friend d_Date           operator-(const d_Date &L, const d_Interval &R);
friend d_Boolean        operator==(const d_Date &L, const d_Date &R);
friend d_Boolean        operator!= (const d_Date &L, const d_Date &R);
friend d_Boolean        operator<  (const d_Date &L, const d_Date &R);
friend d_Boolean        operator<=(const d_Date &L, const d_Date &R);
friend d_Boolean        operator>  (const d_Date &L, const d_Date &R);
friend d_Boolean        operator>=(const d_Date &L, const d_Date &R);
```

```
    d_Boolean       is_between(const d_Date &, const d_Date &) const;
friend d_Boolean    overlaps(const d_Date &psL, const d_Date &peL,
                             const d_Date &psR, const d_Date &peR);
friend d_Boolean    overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                             const d_Date &sR, const d_Date &eR);
friend d_Boolean    overlaps(const d_Date &sL, const d_Date &eL,
                             const d_Timestamp &sR, const d_Timestamp &eR);
static int          days_in_year(unsigned short year);
    int             days_in_year() const;
static int          days_in_month(unsigned short yr, unsigned short month);
    int             days_in_month() const;
static d_Boolean    is_valid_date(unsigned short year, unsigned short month,
                                  unsigned short day);
};
```

If an attempt is made to set a d_Date object to an invalid value, a d_Error exception object of kind d_Error_DateInvalid is thrown and the value of the d_Date object is undefined.

The functions next, previous, operator+=, and operator–= alter the object and return a reference to the current object. The post increment and decrement operators return a new object by value.

The overlaps functions take two periods (start and end), each period denoted by a start and end time, and determines whether the two time periods overlap. The is_between function determines whether the d_Date value is within a given period.

5.2.1.5 d_Time

The d_Time class is used to denote a specific time, which is internally stored in Greenwich Mean Time (GMT). Initialization, assignment, arithmetic, and comparison operators are defined. There are also functions to access each of the components of a time value. Implementations may have additional functions available to support converting to and from the type used by the operating system to represent a time.

The enumeration Time_Zone is made available to denote a specific time zone. Time zones are numbered according to the number of hours that must be added or subtracted from local time to get the time in Greenwich, England (GMT). Thus the value of GMT is 0. A Time_Zone name of GMT6 indicates a time of 6 hours greater than GMT, and thus 6 must be subtracted from it to get GMT. Conversely, GMT_8 means that the time is 8 hours earlier than GMT (read the underscore as a minus). A default time zone value is maintained and is initially set to the local time zone. It is possible to change the default time zone value as well as reset it to the local value.

Definition:

```
class d_Time {
public:
    enum Time_Zone {
        GMT   = 0,   GMT12 = 12,     GMT_12 = –12,
        GMT1 = 1,    GMT_1 = –1,     GMT2 = 2,       GMT_2 = –2,
        GMT3 = 3,    GMT_3 = –3,     GMT4 = 4,       GMT_4 = –4,
        GMT5 = 5,    GMT_5 = –5,     GMT6 = 6,       GMT_6 = –6,
        GMT7 = 7,    GMT_7 = –7,     GMT8 = 8,       GMT_8 = –8,
        GMT9 = 9,    GMT_9 = –9,     GMT10 = 10,     GMT_10 = –10,
        GMT11 = 11, GMT_11 = –11,
        USeastern = –5,   UScentral = –6, USmountain = –7, USpacific = –8
    };
static  void            set_default_Time_Zone(Time_Zone);
static  void            set_default_Time_Zone_to_local();
                        d_Time();
                        d_Time(unsigned short hour,
                                unsigned short minute, float sec = 0.0f);
                        d_Time(unsigned short hour, unsigned short minute,
                                float sec, short tzhour, short tzminute);
                        d_Time(const d_Time &);
                        d_Time(const d_Timestamp &);
    d_Time &            operator=(const d_Time &);
    d_Time &            operator=(const d_Timestamp &);
    unsigned short      hour() const;
    unsigned short      minute() const;
    Time_Zone           time_zone() const;
    float               second() const;
    short               tz_hour() const;
    short               tz_minute() const;
static  d_Time          current();
    d_Time &            operator+=(const d_Interval &);
    d_Time &            operator–=(const d_Interval &);
friend  d_Time          operator+(const d_Time &L, const d_Interval &R);
friend  d_Time          operator+(const d_Interval &L, const d_Time &R);
friend  d_Interval      operator–(const d_Time &L, const d_Time &R);
friend  d_Time          operator–(const d_Time &L, const d_Interval &R);
friend  d_Boolean       operator==(const d_Time &L, const d_Time &R);
friend  d_Boolean       operator!= (const d_Time &L, const d_Time &R);
friend  d_Boolean       operator<  (const d_Time &L, const d_Time &R);
```

```
friend  d_Boolean     operator<=(const d_Time &L, const d_Time &R);
friend  d_Boolean     operator>  (const d_Time &L, const d_Time &R);
friend  d_Boolean     operator>=(const d_Time &L, const d_Time &R);
friend  d_Boolean     overlaps(const d_Time &psL, const d_Time &peL,
                               const d_Time &psR, const d_Time &peR);
friend  d_Boolean     overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                               const d_Time &sR, const d_Time &eR);
friend  d_Boolean     overlaps(const d_Time &sL, const d_Time &eL,
                               const d_Timestamp &sR, const d_Timestamp &eR);
};
```

All arithmetic on d_Time is done on a modulo 24-hour basis. If an attempt is made to set a d_Time object to an invalid value, a d_Error exception object of kind d_Error_TimeInvalid is thrown and the value of the d_Time object is undefined.

The default d_Time constructor initializes the object to the current time. The overlaps functions take two periods, each denoted by a start and end time, and determines whether the two time periods overlap.

5.2.1.6 d_Timestamp

A d_Timestamp consists of both a date and time.

Definition:

```
class d_Timestamp {
public:
                        d_Timestamp();     // sets to the current date/time
                        d_Timestamp(unsigned short year, unsigned short month=1,
                                unsigned short day = 1, unsigned short hour = 0,
                                unsigned short minute = 0, float sec = 0.0);
                        d_Timestamp(const d_Date &);
                        d_Timestamp(const d_Date &, const d_Time &);
                        d_Timestamp(const d_Timestamp &);
    d_Timestamp &       operator=(const d_Timestamp &);
    d_Timestamp &       operator=(const d_Date &);
    const d_Date &      date() const;
    const d_Time &      time() const;
    unsigned short      year() const;
    unsigned short      month() const;
    unsigned short      day() const;
    unsigned short      hour() const;
    unsigned short      minute() const;
    float               second() const;
    short               tz_hour() const;
    short               tz_minute() const;
```

```
static  d_Timestamp  current();
        d_Timestamp &  operator+=(const d_Interval &);
        d_Timestamp &  operator-=(const d_Interval &);
friend  d_Timestamp  operator+(const d_Timestamp &L, const d_Interval &R);
friend  d_Timestamp  operator+(const d_Interval &L, const d_Timestamp &R);
friend  d_Timestamp  operator-(const d_Timestamp &L, const d_Interval &R);
friend  d_Interval   operator-(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator==(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator!= (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator<  (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator<=(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator>  (const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    operator>=(const d_Timestamp &L, const d_Timestamp &R);
friend  d_Boolean    overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                              const d_Timestamp &sR, const d_Timestamp &eR);
friend  d_Boolean    overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                              const d_Date &sR, const d_Date &eR);
friend  d_Boolean    overlaps(const d_Date &sL, const d_Date &eL,
                              const d_Timestamp &sR, const d_Timestamp &eR);
friend  d_Boolean    overlaps(const d_Timestamp &sL, const d_Timestamp &eL,
                              const d_Time &sR, const d_Time &eR);
friend  d_Boolean    overlaps(const d_Time &sL, const d_Time &eL,
                              const d_Timestamp &sR, const d_Timestamp &eR);
};
```

If an attempt is made to set the value of a d_Timestamp object to an invalid value, a d_Error exception object of kind d_Error_TimestampInvalid is thrown and the value of the d_Timestamp object is undefined.

5.2.2 Relationship Traversal Path Declarations

Relationships do not have syntactically separate definitions. Instead, the *traversal paths* used to cross relationships are defined within the bodies of the definitions of each of the two object types that serve a role in the relationship. For example, if there is a one-to-many relationship between professors and the students they have as advisees, then the traversal path advisees is defined within the type definition of the object type Professor**,** and the traversal path advisor is defined within the type definition of the object type Student.

A relationship traversal path declaration is similar to an attribute declaration, but with the following differences. Each end of a relationship has a relationship traversal path. A traversal path declaration is an attribute declaration and must be of type

- d_Rel_Ref<T, const char *> (which has the interface of d_Ref<T>)
- d_Rel_Set<T, const char *> (which has the interface of d_Set<d_Ref<T> >)
- d_Rel_List<T, const char *> (which has the interface of d_List<d_Ref<T> >)

for some persistent class T. The second template argument should be a variable that contains the name of the attribute in the other class, which serves as the inverse role in the relationship. Both classes in a relationship must have a member of one of these types, and the members of the two classes must refer to each other. If the second template argument has a name that does not correspond to a data member in the referenced class, a d_Error exception object of kind d_Error_MemberNotFound is thrown. If the second template argument does refer to a data member, but the data member is the wrong type, i.e., it is not of type d_Rel_Ref, d_Rel_Set, or d_Rel_List, a d_Error exception object of kind d_Error_MemberIsOfInvalidType is thrown.

Studying the relationships in the examples below will make this clear.

Examples:

```
extern const char _dept [ ],       _professors [ ] ;
extern const char _advisor [ ],    _advisees [ ] ;
extern const char _classes [ ],    _enrolled [ ] ;

class Department : public d_Object {
public:
    d_Rel_Set<Professor, _dept>             professors;
};
class Professor : public d_Object {
public:
    d_Rel_Ref<Department, _professors>      dept;
    d_Rel_Set<Student, _advisor>            advisees;
};
class Student : public d_Object {
public:
    d_Rel_Ref<Professor, _advisees>         advisor;
    d_Rel_Set<Course, _enrolled>            classes;
};
class Course : public d_Object {
public:
    d_Rel_Set<Student, _classes>            students_enrolled;
};
const char _dept [ ] = "dept";
const char _professors [ ] = "professors";
```

```
const char _advisor [ ] = "advisor";
const char _advisees [ ] = "advisees";
const char _classes [ ] = "classes" ;
const char _enrolled [ ] = "students_enrolled";
```

The second template parameter is based on the address of the variable, not on the string contents. Thus a different variable is required for each role, even if the member name happens to be the same. The string contents must match the name of the member in the other class involved in the relationship.

The referential integrity of bidirectional relationships is automatically maintained. If a relationship exists between two objects and one of the objects gets deleted, the relationship is considered to no longer exist and the inverse traversal path will be altered to remove the relationship.

5.2.3 Operation Declarations

Operation declarations in C++ are syntactically identical to *function member* declarations. For example, see grant_tenure and assign_course defined for class Professor in Section 5.2.

5.3 C++ OML

This section describes the C++ binding for the OML. A guiding principle in the design of C++ OML is that the syntax used to create, delete, identify, reference, get/set property values, and invoke operations on a persistent object should be, so far as possible, no different than that used for objects of shorter lifetimes. A single expression may freely intermix references to persistent and transient objects.

While it is our long-term goal that nothing can be done with persistent objects that cannot also be done with transient objects, this standard treats persistent and transient objects slightly differently. Queries and transaction consistency apply only to persistent objects.

5.3.1 Object Creation, Deletion, Modification, and References

Objects can be created, deleted, and modified. Objects are created in C++ OML using the new operator, which is overloaded to accept additional arguments specifying the lifetime of the object. An optional storage pragma allows the programmer to specify how the newly allocated object is to be clustered with respect to other objects.

The static member variable d_Database::transient_memory is defined in order to allow libraries that create objects to be used uniformly to create objects of any lifetime. This variable may be used as the value of the database argument to operator new to create objects of transient lifetime.

```
static d_Database * const d_Database::transient_memory;
```

The formal ODMG forms of the C++ new operator are

```
(1) void * operator new(size_t size);
(2) void * operator new(size_t size, const d_Ref_Any &clustering,
                        const char* typename);
(3) void * operator new(size_t size, d_Database *database,
                        const char* typename);
```

These operators have d_Object scope. (1) is used for creation of transient objects derived from d_Object. (2) and (3) create persistent objects. In (2) the user specifies that the newly created object should be placed "near" the existing clustering object. The exact interpretation of "near" is implementation-defined. An example interpretation would be "on the same page if possible." In (3) the user specifies that the newly created object should be placed in the specified database, but no further clustering is specified.

The size argument, which appears as the first argument in each signature, is the size of the representation of an object. It is determined by the compiler as a function of the class of which the new object is an instance, not passed as an explicit argument by a programmer writing in the language.

If the database does not have the schema information about a class when new is called, a d_Error exception object of kind d_Error_DatabaseClassUndefined is thrown.

Examples:

```
    d_Database *yourDB, *myDB;  // assume these get initialized properly
(1) d_Ref<Schedule> temp_sched1 = new Schedule;
(2) d_Ref<Professor> prof2 = new(yourDB,"Professor") Professor;
(3) d_Ref<Student> student1 = new(myDB, "Student") Student;
(4) d_Ref<Student> student2 = new(student1, "Student") Student;
(5) d_Ref<Student> temp_student =
                        new(d_Database::transient_memory, "Student") Student;
```

Statement (1) creates a transient object temp_sched1. Statements (2)–(4) create persistent objects. Statement (2) creates a new instance of class Professor in the database yourDB. Statement (3) creates a new instance of class Student in the database myDB. Statement (4) does the same thing, except that it specifies that the new object, student2, should be placed close to student1. Statement (5) creates a transient object temp_student.

5.3.1.1 Object Deletion

Objects, once created, can be deleted in C++ OML using the d_Ref::delete_object member function. Using the delete operator on a pointer to a persistent object will also delete the object, as in standard C++ practice. Deleting an object is permanent, subject to transaction commit. The object is removed from memory and, if it is a persistent object, from the database. The d_Ref instance or pointer still exists in memory, but its

reference value is undefined. An attempt to access the deleted object is implementation defined.

Example:

```
d_Ref<anyType> obj_ref;
 ...   // set obj_ref to refer to a persistent object
obj_ref.delete_object();
```

C++ requires the operand of delete to be a pointer, so the member function delete_object was defined to delete an object with just a d_Ref<T> reference to it.

5.3.1.2 Object Modification

The state of an object is modified by updating its properties or by invoking operations on it. Updates to persistent objects are made visible to other users of the database when the transaction containing the modifications commits.

Persistent objects that will be modified must communicate to the runtime ODBMS process the fact that their states will change. The ODBMS will then update the database with these new states at transaction commit time. Object change is communicated by invoking the d_Object::mark_modified member function, which is defined in Section 5.3.4 and is used as follows:

```
obj_ref–>mark_modified();
```

The mark_modified function call is included in the constructor and destructor methods for persistence-capable classes, i.e., within class d_Object. The developer should include the call in any other methods that modify persistent objects, before the object is actually modified.

As a convenience, the programmer may omit calls to mark_modified on objects where classes have been compiled using an optional C++ OML preprocessor switch; the system will automatically detect when the objects are modified. In the default case, mark_modified calls are required, because in some ODMG implementations performance will be better when the programmer explicitly calls mark_modified. However, each time a persistent object is modified by a member update function provided explicitly by the ODMG classes, the mark_modified call is not necessary since it is done automatically.

5.3.1.3 Object References

Objects, whether persistent or not, may refer to other objects via object references. In C++ OML object references are instances of the template class d_Ref<T> (see Section 5.3.5). All accesses to persistent objects are made via methods defined on classes d_Ref, d_Object, and d_Database. The dereference operator –> is used to access

members of the persistent object "addressed" by a given object reference. How an object reference is converted to a C++ pointer to the object is implementation-defined.

A dereference operation on an object reference always guarantees that the object referred to is returned or a d_Error exception object of kind d_Error_RefInvalid is thrown. The behavior of a reference is as follows. If an object reference refers to a persistent object that exists but is not in memory when a dereference is performed, it will be retrieved automatically from disk, mapped into memory, and returned as the result of the dereference. If the referenced object does not exist, a d_Error exception object of kind d_Error_RefInvalid is thrown. References to transient objects work exactly the same (at least on the surface) as references to persistent objects.

Any object reference may be set to a null reference or *cleared* to indicate the reference does not refer to an object.

The rules for when an object of one lifetime may refer to an object of another lifetime are a straightforward extension of the C++ rules for its two forms of transient objects—procedure coterminus and process coterminus. An object can always refer to another object of longer lifetime. An object can only refer to an object of shorter lifetime as long as the shorter-lived object exists.

A persistent object is retrieved from disk upon activation. It is the application's responsibility to initialize the values of any of that object's pointers to transient objects. When a persistent object is committed, the ODBMS sets its embedded d_Refs to transient objects to the null value.

5.3.1.4 Object Names

A database application generally will begin processing by accessing one or more critical objects and proceeding from there. These objects are in some sense "root" objects, in that they lead to interconnected webs of other objects. The ability to name an object and retrieve it later by that name facilitates this start-up capability. Named objects are also convenient in many other situations.

There is a single, flat name scope per database; thus all names in a particular database are unique. A name is not explicitly defined as an attribute of an object. The operations for manipulating names are defined in the d_Database class in Section 5.3.8.

5.3.2 Properties

5.3.2.1 Attributes

C++ OML uses standard C++ for accessing attributes. For example, assume prof has been initialized to reference a professor and we wish to modify its id_number:

```
prof–>id_number = next_id;
cout << prof–>id_number;
```

Modifying an attribute's value is considered a modification to the enclosing object instance. One must call mark_modified for the object before it is modified.

The C++ binding allows persistence-capable classes to embed instances of C++ classes, including other persistence-capable classes. However, embedded objects are not considered "independent objects" and have no object identity of their own. Users are not permitted to get a d_Ref to an embedded object. Just as with any attribute, modifying an embedded object is considered a modification to the enclosing object instance, and mark_modified for the enclosing object must be called before the embedded object is modified.

5.3.2.2 Relationships

The ODL specifies which relationships exist between object classes. Creating, traversing, and breaking relationships between instances are defined in the C++ OML. Both to-one and to-many traversal paths are supported by the OML. The integrity of relationships is maintained by the ODBMS.

The following diagrams will show graphically the effect of adding, modifying, and deleting relationships among classes. Each diagram is given a name to reflect the cardinality and resulting effect on the relationship. The name will begin with 1-1, 1-m, or m-m to denote the cardinality and will end in either N (no relationship), A (add a relationship), or M (modify a relationship). When a relationship is deleted, this will result in a state of having no relationship (N). A solid line is drawn to denote the explicit operation performed by the program, and a dashed line shows the side effect operation performed automatically by the ODBMS to maintain referential integrity.

The following template class allows one to specify a to-one relationship to a class T.

```
template <class T, const char *Member> class d_Rel_Ref : public d_Ref<T> {  };
```

The template d_Rel_Ref<T,M> supports the same interface as d_Ref<T>. Implementations will redefine some functions to provide support for referential integrity.

The application programmer must introduce two const char * variables, one used at each end of the relationship to refer to the other end of the relationship, thus establishing the association of the two ends of the relationship. The variables must be initialized with the name of the attribute at the other end of the relationship.

Assume the following 1-1 relationship exists between class A and class B:

```
extern const char _ra [ ], _rb [ ] ;
class A {
      d_Rel_Ref<B, _ra>     rb;
};
```

```
class B {
      d_Rel_Ref<A, _rb>     ra;
};
const char _ra [ ] = "ra";
const char _rb [ ] = "rb";
```

Note that class A and B could be the same class, as well. In each of the diagrams below, there will be an instance of A called a or aa and an instance of B called b or bb. In the following scenario 1-1N there is no relationship between a and b.

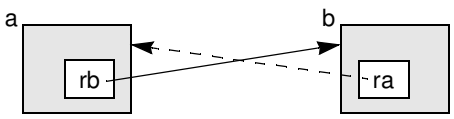
1-1N: No relationship

Then, adding a relationship between a and b via

```
a.rb = &b;
```

results in the following:

1-1A: Add a relationship



The solid arrow indicates the operation specified by the program, and the dashed line shows what operation gets performed automatically by the ODBMS.

Assume now the previous diagram (1-1A) represents the current state of the relationship between a and b. If the program executes the statement:

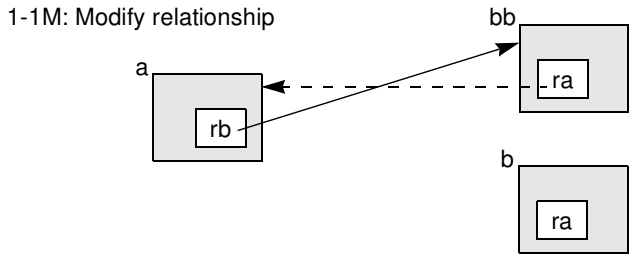
```
a.rb.clear ();
```

the result will be no relationship, as shown in 1-1N.

Assume we have the relationship depicted in 1-1A. If we now execute

```
a.rb = &bb;
```

we obtain the following:



Notice that b.ra no longer refers to A and bb.ra is set automatically to reference a.

Whenever the operand to initialization or assignment represents a null reference, the result will be no relationship as in 1-1N. In the case of assignment, if there had been a relationship, it is removed. If the relationship is currently null (is_null would return true), then doing an assignment would add a relationship, unless the assignment operand was null as well.

If there is currently a relationship with an object, then doing the assignment will modify the relationship as in 1-1M. If the assignment operand is null, then the existing relationship is removed.

When an object involved in a relationship is deleted, all the relationships that the object was involved in will be removed as well.

There are two other cardinalities to consider: one-to-many and many-to-many. With one-to-many and many-to-many relationships, the set of operations allowed are based upon whether the relationship is an unordered set or positional.

The following template class allows one to specify an unordered to-many relationship with a class T:

```
template <class T, const char *M> class d_Rel_Set : public d_Set<d_Ref<T> > { }
```

The template d_Rel_Set<T,M> supports the same interface as d_Set<d_Ref<T> >. Implementations will redefine some functions in order to support referential integrity.

Assuming an unordered one-to-many set relationship between class A and class B:

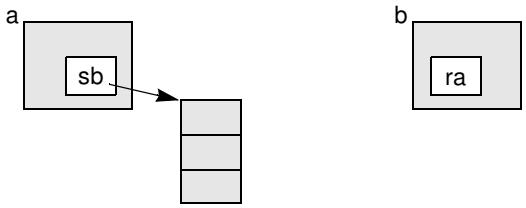
```
extern const char _ra [ ], _sb [ ] ;

class A {
      d_Rel_Set<B, _ra>     sb;
};
class B {
      d_Rel_Ref<A, _sb>     ra;
};
```

```
const char _ra[ ] = "ra";
const char _sb[ ] = "sb";
```

Assume we have the following instances a and b with no relationship.

1-mN: No relationship



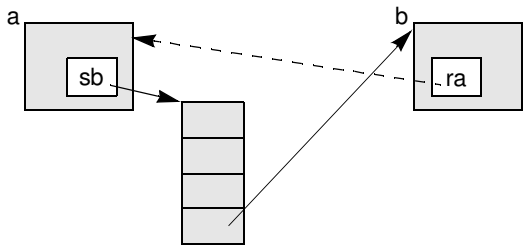
a.sb has 3 elements, but they are referring to instances of B other than b.

Now suppose we add a relationship between a and b by executing the statement

```
a.sb.insert_element (&b);
```

This results in the following:

1-mA: Add a relationship



The b.ra traversal path gets set automatically to reference a. Conversely, if we execute the statement

```
b.ra = &a;
```

an element would have automatically been added to a.sb to refer to b. But only one of the two operations needs to be performed by the program; the ODBMS automatically updates the inverse traversal path.

Given the situation depicted in 1-mA, if we execute either

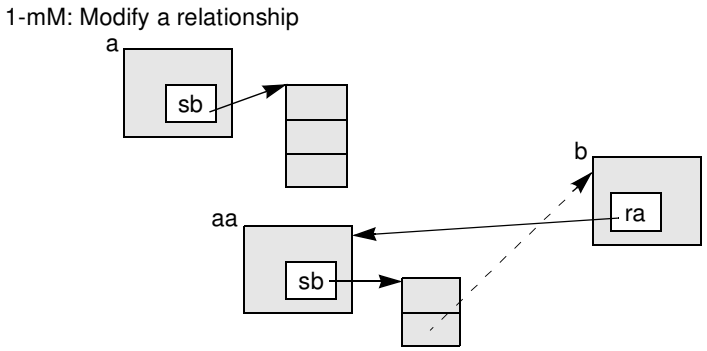
```
a.sb.remove_element (&b)        or        b.ra.clear ();
```

the result would be that the relationship between a and b would be deleted and the state of a and b would be as depicted in 1-mN.

Now assume we have the relationship between a and b as shown in 1-mA. If we execute the following statement:

```
b.ra = &aa;
```

this results in the following:



After the statement executes, b.ra refers to aa, and as a side effect, the element within a.sb that had referred to b is removed and an element is added to aa.sb to refer to b.

The d_List class represents a *positional* collection, whereas the d_Set class is an unordered collection. Likewise, the d_Rel_List<T, Member> template is used for representing relationships that are positional in nature.

```
template <class T, const char *M> class d_Rel_List : public d_List<d_Ref<T> > { };
```

The template d_Rel_List<T,M> has the same interface as d_List<d_Ref<T> >.

Assuming a positional to-many relationship between class A and class B:

```
extern const char _ra [ ] , _listB [ ] ;

class A {
      d_Rel_List<B, _ra>          listB;
};
class B {
      d_Rel_Ref<A, _listB>        ra;
};
const char _ra [ ] = "ra";
const char _listB [ ] = "listB";
```

The third relationship cardinality to consider is many-to-many. Suppose we have the following relationship between A and B:

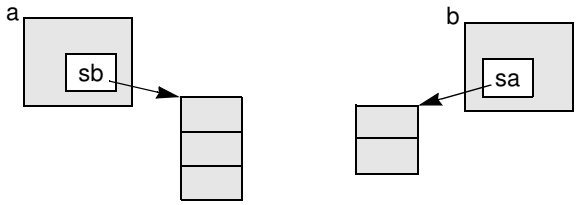
```
extern const char _sa [ ], _sb [ ] ;

class A {
      d_Rel_Set<B, _sa>     sb;
};
```

```
class B {
      d_Rel_Set<A, _sb>     sa;
};
const char _sa [ ] = "sa";
const char _sb [ ] = "sb";
```

Initially, there will be no relationship between instances a and b though a and b have relationships with other instances.

m-mN: No relationship

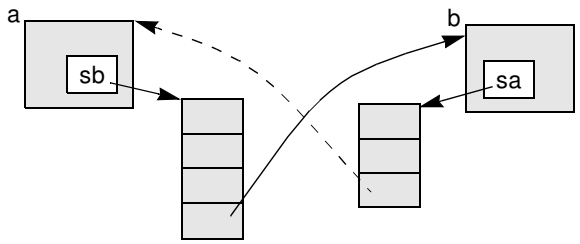


The following statement will add a relationship between a and b:

```
a.sb.insert_element(&b);
```

This will result in the following:

m-mA: Add a relationship



In addition to an element being added to a.sb to reference b, there is an element automatically added to b.sa that references a.

Executing either

```
a.sb.remove_element(&b)
```

or

```
b.sa.remove_element(&a)
```

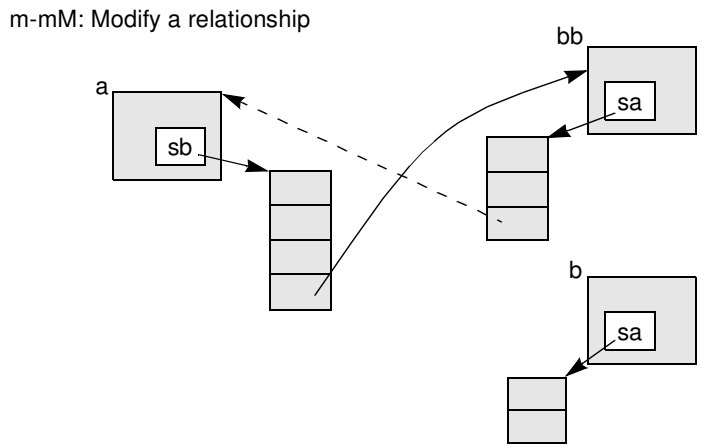
would result in the relationship being removed between a and b, and the result would be as depicted in m-mN.

Last, we consider the modification of a many-to-many relationship. Assume the prior state is the situation depicted in m-mA, and assume that sb represents a positional rela-

tionship. The following statement will modify an existing relationship that exists between a and b, changing a to be related to bb.

```
a.sb.replace_element_at( &bb, 3);
```

This results in the following object relationships:



The result of this operation is that the element in b.sa that referenced a is removed and an element is added to bb.sa to reference a.

The initializations and assignments that have an argument of type d_Rel_Set<T,Member> or d_Set<d_Ref<T> > are much more involved than the simple diagrams above because they involve performing the corresponding operation for every element of the set versus doing it for just one element. The remove_all function removes every member of the relationship, also removing the back-reference for each referenced member. If the assignment operators have an argument that represents an empty set, the assignment will have the same effect as the remove_all function.

Below are some more examples based on the classes used throughout the chapter.

Examples:

```
d_Ref<Professor> p;
d_Ref<Student> Sam;
d_Ref<Department> english_dept;
// initialize p, Sam, and english_dept references
p–>dept = english_dept;   // create 1:1 relationship
p–>dept.clear();              // clear the relationship
p–>advisees.insert_element(Sam); // add Sam to the set of students that are p's
                                 // advisees; same effect as 'Sam–>advisor = p'
p–>advisees.remove_element(Sam); // remove Sam from the set of students that
                                 // are p's advisees, also clears Sam–>advisor
```

5.3.3 Operations

Operations are defined in the OML as they are generally implemented in C++. Operations on transient and persistent objects behave entirely consistently with the operational context defined by standard C++. This includes all overloading, dispatching, function call structure and invocation, member function call structure and invocation, argument passing and resolution, error handling, and compile time rules.

5.3.4 d_Object Class

The class d_Object is introduced and defined as follows:

Definition:

```
class d_Object {
public:
                        d_Object();
                        d_Object(const d_Object &);
virtual                 ~d_Object();
    d_Object &          operator=(const d_Object &);
    void                mark_modified();  // mark the object as modified
    void *              operator new(size_t size);
    void *              operator new(size_t size, const d_Ref_Any &cluster,
                                const char *typename);
    void *              operator new(size_t size, d_Database *database,
                                const char *typename);
    void                operator delete(void *);
virtual  void           d_activate();
virtual  void           d_deactivate();
};
```

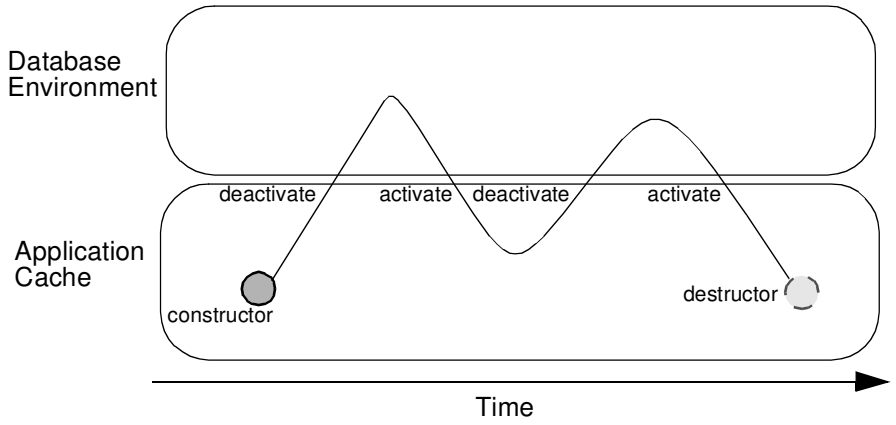
This class is introduced to allow the type definer to specify when a class is capable of having persistent as well as transient instances. Instances of classes derived from d_Object can be either persistent or transient. A class A that is persistence-capable would inherit from class d_Object:

```
class My_Class : public d_Object {...};
```

The delete operator can be used with a pointer to a persistent object to delete the object; the object is removed from both the application cache and the database, which is the same behavior as Ref<T>::delete_object.

An application needs to initialize and manage the transient members of a persistent object as the object enters and exits the application cache. Memory may need to be allocated and deallocated when these events occur, for example. The d_activate function is called when an object enters the application cache, and d_deactivate is called

when the object exits the application cache. Normally C++ code uses the constructor and destructor to perform initialization and destruction, but in an ODMG implementation the constructor only gets called when an object is first created and the destructor is called at the point the object is deleted from the database. The following diagram depicts the calls made throughout the lifetime of an object.



The object first gets initialized by the constructor. At the point the object exits the application cache, the d_deactivate function gets called. When the object reenters the application cache, d_activate gets called. This may get repeated many times, as the object moves in and out of an application cache. Eventually, the object gets deleted, in which case only the destructor gets called, not d_deactivate.

5.3.5 Reference Classes

Objects may refer to other objects through a smart pointer or reference called a d_Ref. A d_Ref<T> is a reference to an instance of type T. There is also a d_Ref_Any class defined that provides a generic reference to any type.

A d_Ref is a template class defined as follows:

Definition:

```
template <class T> class d_Ref {
public:
                        d_Ref();
                        d_Ref(T *fromPtr);
                        d_Ref(const d_Ref<T> &);
                        d_Ref(const d_Ref_Any &);
                        ~d_Ref();
                        operator d_Ref_Any() const;
    d_Ref<T> &          operator=(T *);
    d_Ref<T> &          operator=(const d_Ref<T>&);
```

```
    void            clear();
    T *             operator->() const;    // dereference the reference
    T &             operator*() const;
    T *             ptr() const;
    void            delete_object();  // delete referred object from memory
                                      // and the database
// boolean predicates to check reference
    d_Boolean       operator!() const;
    d_Boolean       is_null() const;
// do these d_Refs and pointers refer to the same objects?
friend d_Boolean    operator==(const d_Ref<T> &refL, const d_Ref<T> &refR );
friend d_Boolean    operator==(const d_Ref<T> &refL, const T *ptrR );
friend d_Boolean    operator==(const T *ptrL, const d_Ref<T> &refR );
friend d_Boolean    operator==(const d_Ref<T> &L, const d_Ref_Any &R);
friend d_Boolean    operator!=(const d_Ref<T> &refL, const d_Ref<T> &refR);
friend d_Boolean    operator!=(const d_Ref<T> &refL, const T *ptrR);
friend d_Boolean    operator!=(const T *ptrL, const d_Ref<T> &refR);
friend d_Boolean    operator!=(const d_Ref<T> &refL, const d_Ref_Any &anyR);
};
```

References in many respects behave like C++ pointers but provide an additional mechanism that guarantees integrity in references to persistent objects. Although the syntax for declaring a d_Ref is different than for declaring a pointer, the usage is, in most cases, the same due to overloading; e.g., d_Refs may be dereferenced with the * operator, assigned with the = operator, etc. A d_Ref to a class may be assigned to a d_Ref to a superclass. d_Refs may be subclassed to provide specific referencing behavior.

There is one anomaly that results from the ability to do conversions between d_Ref<T> and d_Ref_Any. The following code will compile without error, and a d_Error exception object of kind d_Error_TypeInvalid is thrown at run-time versus statically at compile time. Suppose that X and Y are two unrelated classes:

```
d_Ref<X>    x;
d_Ref<Y>    y(x);
```

The initialization of y via x will be done via a conversion to d_Ref_Any. One should avoid such initializations in their application.

The pointer or reference returned by operator–> or operator * is only valid until either the d_Ref is deleted, the end of the outermost transaction, or until the object it points to is deleted. The pointer returned by ptr is only valid until the end of the outermost transaction or until the object it points to is deleted. The value of a d_Ref after a transaction commit or abort is undefined. If an attempt is made to dereference a null

d_Ref<T>, a d_Error exception object of kind d_Error_RefNull is thrown. Calling delete_object with a null d_Ref is silently ignored, as it is with a pointer in C++.

The following template class allows one to specify a to-one relationship to a class T:

```
template <class T, const char *Member> class d_Rel_Ref : public d_Ref<T> {  };
```

The template d_Rel_Ref<T,M> supports the same interface as d_Ref<T>. Implementations will redefine some functions to provide support for referential integrity.

A class d_Ref_Any is defined to support a reference to any type. Its primary purpose is to handle generic references and allow conversions of d_Refs in the type hierarchy. A d_Ref_Any object can be used as an intermediary between any two types d_Ref<X> and d_Ref<Y> where X and Y are different types. A d_Ref<T> can always be converted to a d_Ref_Any; there is a function to perform the conversion in the d_Ref<T> template. Each d_Ref<T> class has a constructor and assignment operator that takes a reference to a d_Ref_Any.

The d_Ref_Any class is defined as follows:

Definition:

```
class d_Ref_Any {
public:
                        d_Ref_Any();
                        d_Ref_Any(const d_Ref_Any &);
                        d_Ref_Any(d_Object *);
                        ~d_Ref_Any();
        d_Ref_Any &     operator=(const d_Ref_Any &);
        d_Ref_Any &     operator=(d_Object *);
        void            clear();
        void            delete_object();

// boolean predicates checking to see if value is null or not
        d_Boolean       operator!() const;
        d_Boolean       is_null() const;

friend  d_Boolean       operator==(const d_Ref_Any &, const d_Ref_Any &);
friend  d_Boolean       operator==(const d_Ref_Any &, const d_Object *);
friend  d_Boolean       operator==(const d_Object *, const d_Ref_Any &);
friend  d_Boolean       operator!=(const d_Ref_Any &, const d_Ref_Any &);
friend  d_Boolean       operator!=(const d_Ref_Any &, const d_Object *);
friend  d_Boolean       operator!=(const d_Object *, const d_Ref_Any &);
};
```

The operations defined on d_Ref<T> that are not dependent on a specific type T have been provided in the d_Ref_Any class.

5.3.6 Collection Classes

Collection templates are provided to support the representation of a collection whose elements are of an arbitrary type. A conforming implementation must support at least the following subtypes of d_Collection:

- d_Set
- d_Bag
- d_List
- d_Varray
- d_Dictionary

The C++ class definitions for each of these types are defined in the subsections that follow. Iterators are defined as a final subsection.

The following discussion uses the d_Set class in its explanation of collections, but the description applies for all concrete classes derived from d_Collection.

Given an object of type T, the declaration

```
d_Set<T> s;
```

defines a d_Set collection whose elements are of type T. If this set is assigned to another set of the same type, both the d_Set object itself and each of the elements of the set are copied. The elements are copied using the copy semantics defined for the type T. A common convention will be to have a collection that contains d_Refs to persistent objects—for example,

```
d_Set<d_Ref<Professor> > faculty;
```

The d_Ref class has shallow copy semantics. For a d_Set<T>, if T is of type d_Ref<C> for some persistence-capable class C, only the d_Ref objects are copied, not the C objects that the d_Ref objects reference.

This holds in any scope; in particular, if s is declared as a member inside a class, the set itself will be embedded inside an instance of this class. When an object of this enclosing class is copied into another object of the same enclosing class, the embedded set is copied, too, following the copy semantics defined above. This must be differentiated from the declaration

```
d_Ref<d_Set<T> >  ref_set;
```

which defines a reference to a d_Set. When such a reference is defined as a property of a class, that means that the set itself is an independent object that lies outside an instance of the enclosing class. Several objects may then share the same set, since copying an object will not copy the set, but just the reference to it. These are illustrated in Figure 5-2.

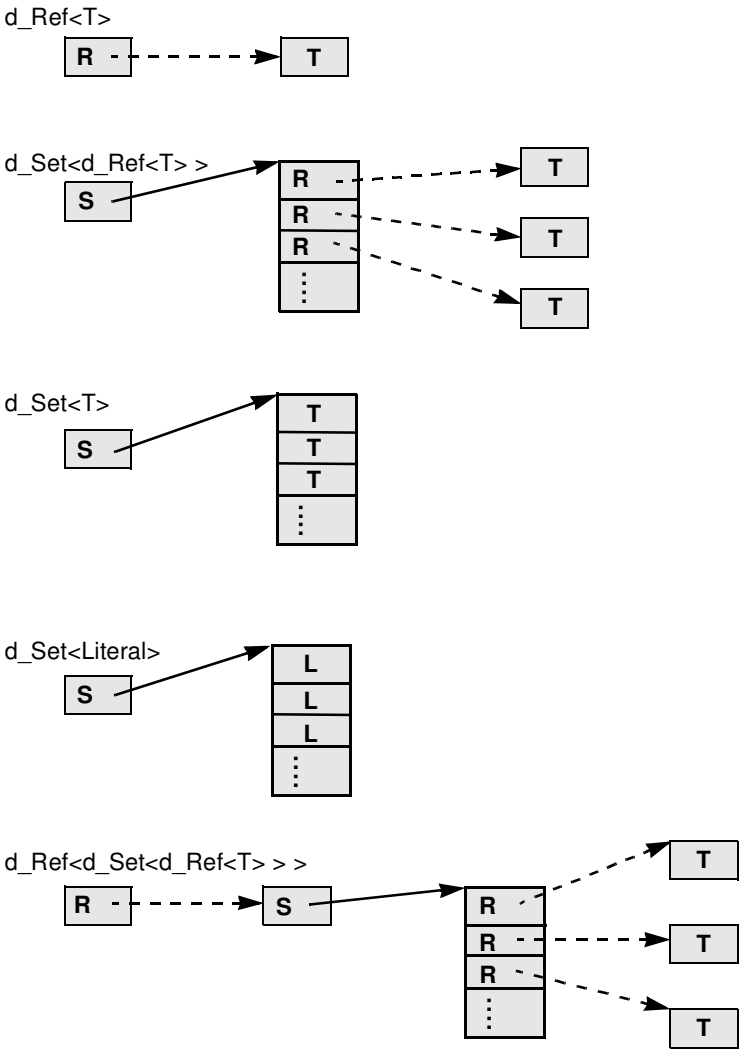


Figure 5-2. Collections, Embedded and with d_Ref

Collection elements may be of any type. Every type T that will become an element of a given collection must support the following operations:

```
class T {
public:
        T();
        T(const T &);
        ~T();
```

```
    T &       operator=(const T &);
friend int    operator==(const T&, const T&);
};
```

This is the complete set of functions required for defining the copy semantics for a given type. For types requiring ordering, the following operation must also be provided:

```
friend d_Boolean      operator<(const T&, const T&);
```

Note that the C++ compiler will automatically generate a copy constructor and assignment operator if the class designer does not declare one. Note that the d_Ref<T> class supports these operations, except for operator<.

Collections of literals, including both atomic and structured literals, are defined as part of the standard. This includes both primitive and user-defined types; e.g., d_Set<int>, d_Set<struct time_t> will be defined with the same behavior.

Figure 5-2 illustrates various types involving d_Sets, d_Refs, a literal type L (int for example) and a persistent class T. The d_Set object itself is represented by a box that then refers to a set of elements of the specified type. A solid arrow is used to denote containment of the set elements within the d_Set object. d_Refs have a dashed arrow pointing to the referenced object of type T.

5.3.6.1 Class d_Collection

Class d_Collection is an abstract class in C++ and cannot have instances. It is derived from d_Object, allowing instances of concrete classes derived from d_Collection to be stand-alone persistent objects.

Definition:

```
template <class T> class d_Collection : public d_Object {
public:
virtual                    ~d_Collection();
    d_Collection<T> &      assign_from(const d_Collection<T> &);
friend d_Boolean           operator==( const d_Collection<T> &cL,
                                       const d_Collection<T> &cR);
friend d_Boolean           operator!=( const d_Collection<T> &cL,
                                       const d_Collection<T> &cR);
    unsigned long          cardinality() const;
    d_Boolean              is_empty() const;
    d_Boolean              is_ordered() const;
    d_Boolean              allows_duplicates() const;
    d_Boolean              contains_element(const T &element) const;
```

```
        void                insert_element(const T &elem);
        void                remove_element(const T &elem);
        void                remove_all();
        void                remove_all(const T &elem);
        d_Iterator<T>       create_iterator() const;
        d_Iterator<T>       begin() const;
        d_Iterator<T>       end() const;
        T                   select_element(const char *OQL_predicate) const;
        d_Iterator<T>       select(const char * OQL_predicate) const;
        int                 query(d_Collection<T> &, const char *OQL_pred) const;
        d_Boolean           exists_element(const char* OQL_predicate) const;
    protected:
                            d_Collection(const d_Collection<T> &);
        d_Collection<T> &   operator=(const d_Collection<T> &);
                            d_Collection();
    };
```

Note that the d_Collection class provides the operation assign_from in place of operator= because d_Collection assignment is relatively expensive. This will prevent the often gratuitous use of assignment with collections.

The member function remove_element() removes one element that is equal to the argument from the collection. For ordered collections this will be the element that is equal to the value that would be first encountered if performing a forward iteration of the collection. The remove_all() function removes all of the elements from the collection. The remove_all(const T&) function removes all of the elements in the collection that are equal to the supplied value.

The destructor is called for an element whenever the element is removed from the collection. This also applies when the collection itself is assigned to or removed. If the element type of the collection is d_Ref<T> for some class T, the destructor of d_Ref<T> is called, but not the destructor of T.

The equality can be evaluated of two instances of any collection class derived from d_Collection that have the same element type. When comparing an instance of d_Set to either an instance of d_Set, d_Bag, d_List, or d_Varray, they are only equal if they have the same cardinality and the same elements. When comparing an instance of d_Bag to an instance of d_Bag, d_List, or d_Varray, they are equal if they have the same cardinality and the same number of occurrences of each element value. The ordering of the elements in the d_List or d_Varray does not matter; they are treated like a d_Bag. An instance of d_List or d_Varray is equal to another instance of d_List or d_Varray if they have the same cardinality and the element at each position is equal.

The create_iterator function returns an iterator pointing at the first element in the collection. The begin and end functions are supplied for compatibility with the C++ Standard Template Library (STL) algorithms. The function begin returns an iterator positioned at the first element of iteration. As defined in STL, the end function returns an iterator value that is "past the end" of iteration and is not dereferenceable.

5.3.6.2 Class d_Set

A d_Set<T> is an unordered collection of elements of type T with no duplicates.

Definition:

```
template <class T> class d_Set : public d_Collection<T> {
public:
                        d_Set();
                        d_Set(const d_Set<T> &);
                        ~d_Set();
      d_Set<T> &        operator=(const d_Set<T> &);
      d_Set<T> &        union_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        union_with(const d_Set<T> &s2);
      d_Set<T> &        operator+=(const d_Set<T> &s2);         // union_with
      d_Set<T>          create_union(const d_Set<T> &s) const;
friend  d_Set<T>        operator+(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Set<T> &        intersection_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        intersection_with(const d_Set<T> &s2);
      d_Set<T> &        operator*=(const d_Set<T> &s2);         // intersection_with
      d_Set<T>          create_intersection(const d_Set<T> &s) const;
friend  d_Set<T>        operator*(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Set<T> &        difference_of(const d_Set<T> &sL, const d_Set<T> &sR);
      d_Set<T> &        difference_with(const d_Set<T> &s2);
      d_Set<T> &        operator-=(const d_Set<T> &s2);         // difference_with
      d_Set<T>          create_difference(const d_Set<T> &s) const;
friend  d_Set<T>        operator-(const d_Set<T> &s1, const d_Set<T> &s2);
      d_Boolean         is_subset_of(const d_Set<T> &s2) const;
      d_Boolean         is_proper_subset_of(const  d_Set<T> &s2) const;
      d_Boolean         is_superset_of(const d_Set<T> &s2) const;
      d_Boolean         is_proper_superset_of(const d_Set<T> &s2) const;
};
```

Note that all operations defined on type d_Collection are inherited by type d_Set, e.g., insert_element, remove_element, select_element, and select.

Examples:

- creation:

```
d_Database db;                  // assume we open a database
d_Ref<Professor> Guttag; // assume we set this to a professor
d_Ref<d_Set<d_Ref<Professor> > > my_profs =
                                        new(&db) d_Set<d_Ref<Professor> >;
```

- insertion:

```
my_profs–>insert_element(Guttag);
```

- removal:

```
my_profs–>remove_element(Guttag);
```

- deletion:

```
my_profs.delete_object();
```

For each of the set operations (union, intersection, and difference) there are three ways of computing the resulting set. These will be explained using the union operation. Each one of the union functions has two set operands and computes their union. They vary in how the set operands are passed and how the result is returned, to support different interface styles. The union_of function is a member function that has two arguments, which are references to d_Set<T>. It computes the union of the two sets and places the result in the d_Set object with which the function was called, removing the original contents of the set. The union_with function is also a member and places its result in the object with which the operation is invoked, removing its original contents. The difference is that union_with uses its current set contents as one of the two operands being unioned, thus requiring only one operand passed to the member function. Both union_of and union_with return a reference to the object with which the operation was invoked. The union_with function has a corresponding operator+= function defined. On the other hand, create_union creates and returns a new d_Set instance by value that contains the union, leaving the two original sets unaltered. This function also has a corresponding operator+ function defined.

The following template class allows one to specify an unordered to-many relationship with a class T:

```
template <class T, const char *M> class d_Rel_Set : public d_Set<d_Ref<T> > { }
```

The template d_Rel_Set<T,M> supports the same interface as d_Set<d_Ref<T> >. Implementations will redefine some functions in order to support referential integrity.

5.3.6.3 Class d_Bag

A d_Bag<T> is an unordered collection of elements of type T that does allow for duplicate values.

Definition:

```
template <class T> class d_Bag : public d_Collection<T> {
public:
                        d_Bag();
                        d_Bag(const d_Bag<T> &);
                        ~d_Bag();
    d_Bag<T> &          operator=(const d_Bag<T> &);
    unsigned long       occurrences_of(const T &element) const;
    d_Bag<T> &          union_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          union_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator+=(const d_Bag<T> &b2); // union_with
    d_Bag<T>            create_union(const d_Bag<T> &b) const;
friend  d_Bag<T>        operator+(const d_Bag<T> &b1, const d_Bag<T> &b2);
    d_Bag<T> &          intersection_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          intersection_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator*=(const d_Bag<T> &b2);   // intersection_with
    d_Bag<T>            create_intersection(const d_Bag<T>&b) const;
friend  d_Bag<T>        operator*(const d_Bag<T> &b1, const d_Bag<T> &b2);
    d_Bag<T> &          difference_of(const d_Bag<T> &bL, const d_Bag<T> &bR);
    d_Bag<T> &          difference_with(const d_Bag<T> &b2);
    d_Bag<T> &          operator-=(const d_Bag<T> &b2); // difference_with
    d_Bag<T>            create_difference(const d_Bag<T> &b) const;
friend  d_Bag<T>        operator-(const d_Bag<T> &b1, const d_Bag<T> &b2);
};
```

The union, intersection, and difference operations are described in the section above on the d_Set class.

5.3.6.4 Class d_List

A d_List<T> is an ordered collection of elements of type T and does allow for duplicate values. The beginning d_List index value is 0, following the convention of C and C++.

Definition:

```
template <class T> class d_List : public d_Collection<T> {
public:
                        d_List();
                        d_List(const d_List<T> &);
                        ~d_List();
```

```
    d_List<T> &     operator=(const d_List<T> &);
    T               retrieve_first_element() const;
    T               retrieve_last_element() const;
    void            remove_first_element();
    void            remove_last_element();
    T               operator[](unsigned long position) const;
    d_Boolean       find_element(const T &element,
                                 unsigned long &position) const;
    T               retrieve_element_at(unsigned long position) const;
    void            remove_element_at(unsigned long position);
    void            replace_element_at(const T &element,
                                       unsigned long position);
    void            insert_element_first(const T &element);
    void            insert_element_last(const T &element);
    void            insert_element_after(const T & element,
                                       unsigned long position);
    void            insert_element_before(const T &element,
                                         unsigned long position);
    d_List<T>       concat(const d_List<T> &listR) const;
friend  d_List<T>   operator+(const d_List<T> &listL, const d_List<T> &listR);
    d_List<T> &     append(const d_List<T> &listR);
    d_List<T> &     operator+=(const d_List<T> &listR);
};
```

The insert_element function (inherited from d_Collection <T>) inserts a new element at the end of the list. The subscript operator (operator[]) has the same semantics as the member function retrieve_element_at. The concat function creates a new d_List<T> that contains copies of the elements from the original list, followed by copies of the elements from listR. The original lists are not affected. Similarly, operator+ creates a new d_List<T> that contains copies of the elements in listL and listR and does not change either list. The append function and operator+= both copy elements from listR and adds them after the last element of the list. The modified list is returned as the result.

The d_Rel_List<T, Member> template is used for representing relationships that are positional in nature:

```
template <class T, const char *M> class d_Rel_List : public d_List<d_Ref<T> > { };
```

The template d_Rel_List<T,M> has the same interface as d_List<d_Ref<T> >.

5.3.6.5 Class Array

The Array type defined in Section 2.3.6 is implemented by the built-in array defined by the C++ language. This is a single-dimension, fixed-length array.

5.3.6.6 Class d_Varray

A d_Varray<T> is a one-dimensional array of varying length containing elements of type T. The beginning d_Varray index value is 0, following the convention of C and C++.

Definition:

```
template <class T> class d_Varray : public d_Collection<T> {
public:
                          d_Varray();
                          d_Varray(unsigned long length);
                          d_Varray(const d_Varray<T> &);
                          ~d_Varray();
    d_Varray<T> &         operator= (const d_Varray<T> &);
    void                  resize(unsigned long length);
    T                     operator[](unsigned long index) const;
    d_Boolean             find_element(const T &element,
                                       unsigned long &index) const;
    T                     retrieve_element_at(unsigned long index) const;
    void                  remove_element_at(unsigned long index);
    void                  replace_element_at(const T &element,
                                             unsigned long index);
};
```

The insert_element function (inherited from d_Collection <T>) inserts a new element by increasing the d_Varray length by one and placing the new element at this new position in the d_Varray.

Examples:

```
d_Varray<d_Double>  vector(1000);
vector.replace_element_at(3.14159, 97);
vector.resize(2000);
```

5.3.6.7 Class d_Dictionary

The d_Dictionary<K,V> class is an unordered collection of key-value pairs, with no duplicate keys. A key-value pair is represented by an instance of d_Association<K,V>.

```
template <class K, class V> class d_Association
{
public:
    K       key;
    V       value;
            d_Association(const K &k, const V &v) : key(k), value(v) { }
};
```

The d_Dictionary<K,V> inherits from class d_Collection<T> and thus supports all of its base class operations. The insert_element, remove_element, and contains_element operations inherited from d_Collection<T> are valid for d_Dictionary<K,V> types when a d_Association<K,V> is specified as the argument. The contains_element function returns true if both the key and value specified in the d_Association parameter are contained in the dictionary.

```
template <class K, class V>
class d_Dictionary : public d_Collection<d_Association<K,V> > {
public:
                            d_Dictionary();
                            d_Dictionary(const d_Dictionary<K,V> &);
                            ~d_Dictionary();
    d_Dictionary<K,V> &     operator=(const d_Dictionary<K,V> &);
    void                    bind(const K&, const V&);
    void                    unbind(const K&);
    V                       lookup(const K&) const;
    d_Boolean               contains_key(const K&) const;
};
```

Iterating over a d_Dictionary<K,V> object will result in the iteration over a sequence of d_Association<K,V> instances. Each get_element operation, executed on an instance of d_Iterator<T>, returns an instance of d_Association<K,V>. If insert_element inherited from d_Collection is called and a duplicate key is found, its value is replaced with the new value passed to insert_element. The bind operation works the same as insert_element except the key and value are passed separately. When remove_element inherited from d_Collection is called, both the key and value must be equal for the element to be removed. An exception error of d_Error_ElementNotFound is thrown if the d_Association is not found in the dictionary. The function unbind removes the element with the specified key.

5.3.6.8 Class d_Iterator

A template class, d_Iterator<T>, defines the generic behavior for iteration. All iterators use a consistent protocol for sequentially returning each element from the collection

over which the iteration is defined. A template class has been used to give us type-safe iterators, i.e., iterators that are guaranteed to return an instance of the type of the element of the collection over which the iterator is defined. Normally, an iterator is initialized by the create_iterator method on a collection class.

The template class d_Iterator<T> is defined as follows:

```
template <class T> class d_Iterator {
public:
                          d_Iterator();
                          d_Iterator(const d_Iterator<T> &);
                          ~d_Iterator();
      d_Iterator<T> &     operator=(const d_Iterator<T> &);
friend  d_Boolean         operator==(const d_Iterator<T> &, const d_Iterator<T> &);
friend  d_Boolean         operator!=(const d_Iterator<T> &, const d_Iterator<T> &);
      void                reset();
      d_Boolean           not_done() const;
      void                advance();
      d_Iterator<T> &     operator++();
      d_Iterator<T>       operator++(int);
      d_Iterator<T> &     operator--();
      d_Iterator<T>       operator--(int);
      T                   get_element() const;
      T                   operator*() const;
      void                replace_element(const T &);
      d_Boolean           next(T &objRef);
};
```

When an iterator is constructed, it is either initialized with another iterator or set to null. When an iterator is constructed via the create_iterator function defined in d_Collection, the iterator is initialized to point to the first element, if there is one. Iterator assignment is also supported. A reset function is provided to reinitialize the iterator to the start of iteration for the same collection. The replace_element function can only be used with d_List or d_Varray.

The not_done function allows one to determine whether there are any more elements in the collection to be visited in the iteration. It returns 1 if there are more elements and 0 if iteration is complete. The advance function moves the iterator forward to the next element in the collection. The prefix and postfix forms of the increment operator ++ have been overloaded to provide an equivalent advance operation. One can also move backward through the collection by using the decrement operator --. However, using the -- decrement operator on an iterator of an unordered collection will throw a d_Error exception object of kind d_Error_IteratorNotBackward. If an attempt is made

to either advance an iterator once it has already reached the end of a collection or move backward once the first element has been reached, a d_Error exception object of kind d_Error_IteratorExhausted is thrown. An attempt to use an iterator with a different collection than the collection it is associated with causes a d_Error exception object of kind d_Error_IteratorDifferentCollections to be thrown.

The get_element function and operator* return the value of the current element. If there is no current element, a d_Error exception object of kind d_Error_IteratorExhausted is thrown. There would be no current element if iteration had been completed (not_done return of 0) or if the collection had no elements.

The next function provides a facility for checking the end of iteration, advancing the iterator and returning the current element, if there is one. Its behavior is as follows:

```
template <class T> d_Boolean d_Iterator<T>::next(T &objRef)
{
    if( !not_done() ) return 0;   // no more elements, return false
    objRef = get_element();       // assign current element into output parameter
    advance();                    // advance to the next element
    return 1;                     // return true, that there is a next element
}
```

These operations allow for two styles of iteration, using either a while or for loop.

Example:

Given the class Student, with extent students:

```
(1) d_Iterator<d_Ref<Student> >  iter = students.create_iterator();
    d_Ref<Student>  s;
(2) while( iter.next(s) ) {
        ....
    }
```

Note that calling get_element after calling next will return a different element (the next element, if there is one). This is due to the fact that next will access the current element and then advance the iterator before returning.

Or equivalently with a for loop:

```
(3) d_Iterator<d_Ref<Student> > iter = students.create_iterator();
(4) for( ; iter.not_done(); ++iter) {
(5)     d_Ref<Student> s = iter.get_element();
        ....
    }
```

Statement (1) defines an iterator iter that ranges over the collection students. Statement (2) iterates through this collection, returning a d_Ref to a Student on each successive call to next, binding it to the loop variable s. The body of the while statement is then executed once for each student in the collection students. In the for loop (3) the iterator is initialized, iteration is checked for completion, and the iterator is advanced. Inside the for loop the get_element function can be called to get the current element.

5.3.6.9 Collections and the Standard Template Library

The C++ Standard Template Library (STL) provides an extensible set of containers, i.e., collections and algorithms that work together in a seamless way. The ODMG C++ language binding extends STL with persistence-capable versions of STL's container classes, each of which may be operated on by all template algorithms in the same manner as transient containers. A conforming implementation must provide at least the following persistence-capable STL container types, derived from d_Object:

- d_set
- d_multiset
- d_vector
- d_list
- d_map
- d_multimap

The names of these containers have the ODMG prefix (d_) and have interfaces that correspond to the STL set, multiset, vector, list, map, and multimap containers, respectively.

The C++ STL may be used to operate on the collection classes derived from d_Collection defined in the C++ OML. STL algorithms traverse container data structures using iterator objects, which are compatible with the d_Iterator<T> objects defined in the C++ OML. Specifically, d_Iterator<T> objects conform to the STL specification of constant iterators of the bidirectional_iterator category, though an implementation may provide more powerful iterators in some circumstances.

5.3.7 Transactions

Transaction semantics are defined in the object model explained in Chapter 2.

Transactions can be started, committed, aborted, and checkpointed. It is important to note that *all access, creation, modification, and deletion of persistent objects must be done within a transaction.*

Transactions are implemented in C++ as objects of class d_Transaction. The class d_Transaction defines the operation for starting, committing, aborting, and checkpointing transactions. These operations are

```
class d_Transaction {
public:
                        d_Transaction();
                        ~d_Transaction();
    void                begin();
    void                commit();
    void                abort();
    void                checkpoint();
// Thread operations
    void                join();
    void                leave();
    d_Boolean           is_active() const;
static d_Transaction *  current();
private:
                        d_Transaction(const d_Transaction &);
    d_Transaction &     operator=(const d_Transaction &);
};
```

Transactions must be explicitly created and started; they are not automatically started on database open, upon creation of a d_Transaction object, or following a transaction commit or abort.

The begin function starts a transaction. Calling begin multiple times on the same transaction object, without an intervening commit or abort, causes a d_Error exception object of kind d_Error_TransactionOpen to be thrown on second and subsequent calls. If a call is made to commit, checkpoint, or abort on a transaction object and a call had not been initially made to begin, a d_Error exception object of kind d_Error_TransactionNotOpen is thrown.

Calling commit commits to the database all persistent objects modified (including those created or deleted) within the transaction and releases any locks held by the transaction. Implementations may choose to maintain the validity of d_Refs to persistent objects across transaction boundaries. The commit operation does not delete the transaction object.

Calling checkpoint commits objects modified within the transaction since the last checkpoint to the database. The transaction retains all locks it held on those objects at the time the checkpoint was invoked. All d_Refs and pointers remain unchanged.

Calling abort aborts changes to objects and releases the locks, and does not delete the transaction object.

The destructor aborts the transaction if it is active.

The boolean function is_active returns d_True if the transaction is active; otherwise it returns d_False.

In the current standard, transient objects are not subject to transaction semantics. Committing a transaction does not remove transient objects from memory. Aborting a transaction does not restore the state of modified transient objects.

d_Transaction objects are not long-lived (beyond process boundaries) and cannot be stored to the database. This means that transaction objects may not be made persistent and that the notion of "long transactions" is not defined in this specification.

In summary the rules that apply to object modification (necessarily, during a transaction) are

1. Changes made to persistent objects within a transaction can be "undone" by aborting the transaction.
2. Transient objects are standard C++ objects.
3. Persistent objects created within the scope of a transaction are handled consistently at transaction boundaries: stored to the database and removed from memory (at transaction commit) or deleted (as a result of a transaction abort).

A thread must explicitly create a transaction object or associate itself with an existing transaction object by calling join. The member function join attaches the caller's thread to the transaction and the thread is detached from any other transaction it may be associated with. Calling begin on a transaction object without doing a prior join implicitly joins the transaction to the calling thread. All subsequent operations by the thread, including reads, writes, and implicit lock acquisitions, are done under the thread's current transaction.

Calling leave detaches the caller's thread from the d_Transaction instance without attaching the thread to another transaction. The static function current can be called to access the current d_Transaction object that the thread is associated with; null is returned if the thread is not associated with a transaction.

If a transaction is associated with multiple threads, all of these threads are affected by any data operations or transaction operations (begin, commit, checkpoint, abort). Concurrency control on data among threads is up to the client program in this case. In contrast, if threads use separate transactions, the database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers must not pass objects from one thread to another running under a different transaction; ODMG does not define the results of doing this.

There are three ways in which threads can be used with transactions:

1. An application program may have exactly one thread doing database operations, under exactly one transaction. This is the simplest case, and it represents the vast majority of database applications today. Other applications on

separate machines or in separate address spaces may access the same database under separate transactions. A thread can create multiple instances of d_Transaction and can alternate between them by calling join.

2. There may be multiple threads, each with its own separate transaction. This is useful for writing a service accessed by multiple clients on a network. The database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers must not pass objects from one thread to another thread running under a different transaction; ODMG does not define the results of doing this.

3. Multiple threads may share one or more transactions. Using multiple threads per transaction is only recommended for sophisticated programming because concurrency control must be performed by the application.

5.3.8 d_Database Operations

There is a predefined type d_Database. It supports the following methods:

```
class d_Database {
public:
static d_Database * const transient_memory;
    enum access_status { not_open, read_write, read_only, exclusive };
                    d_Database();
    void            open(   const char * database_name,
                            access_status status = read_write);
    void            close();
    void            set_object_name(const d_Ref_Any &theObject,
                                    const char* theName);
    void            rename_object(  const char * oldName,
                                    const char * newName);
    d_Ref_Any       lookup_object(const char * name) const;
private:
                    d_Database(const d_Database &);
    d_Database &    operator=(const d_Database &);
};
```

The database object, like the transaction object, is transient. Databases cannot be created programmatically using the C++ OML defined by this standard. Databases must be opened before starting any transactions that use the database, and closed after ending these transactions.

To open a database, use d_Database::open, which takes the name of the database as its argument. This initializes the instance of the d_Database object.

```
database->open("myDB");
```

Method open locates the named database and makes the appropriate connection to the database. You must open a database before you can access objects in that database. Attempts to open a database when it has already been opened will result in the throwing of a d_Error exception object of kind d_Error_DatabaseOpen. Extensions to the open method will enable some ODBMSs to implement default database names and/or implicitly open a default database when a database session is started. Some ODBMSs may support opening logical as well as physical databases. Some ODBMSs may support being connected to multiple databases at the same time.

To close a database, use d_Database::close:

```
database->close();
```

Method close does appropriate clean-up on the named database connection. After you have closed a database, further attempts to access objects in the database will cause a d_Error exception object of kind d_Error_DatabaseClosed to be thrown. The behavior at program termination if databases are not closed or transactions are not committed or aborted is undefined.

The *name* methods allow manipulating names of objects. The set_object_name method assigns a character string name to the object referenced. If the string supplied as the name argument is not unique within the database, a d_Error exception object of kind d_Error_NameNotUnique will be thrown. Each call to set_object_name for an object adds an additional name to the object. If a value of 0 is passed as the second parameter to set_object_name, all of the names associated with the object are removed.

The rename_object method changes the name of an object. If the new name is already in use, a d_Error exception object of kind d_Error_NameNotUnique will be thrown and the old name is retained. A name can be removed by passing 0 as the second parameter to rename_object. Names are not automatically removed when an object is deleted. If a call is made to lookup_object with the name of a deleted object, a null d_Ref_Any is returned. Previous releases removed the names when the object was deleted.

An object is accessed by name using the d_Database::lookup_object member function. A null d_Ref_Any is returned if no object with the name is found in the database.

Example:

```
d_Ref<Professor> prof = myDatabase->lookup_object("Newton");
```

If a Professor instance named “Newton” exists, it is retrieved and a d_Ref_Any is returned by lookup_object. The d_Ref_Any return value is then used to initialize prof. If the object named “Newton” is not an instance of Professor or a subclass of Professor, a d_Error exception object of kind d_Error_TypeInvalid is thrown during this initialization.

If the definition of a class in the application does not match the database definition of the class, a d_Error exception object of kind d_Error_DatabaseClassMismatch is thrown.

5.3.9 Class d_Extent<T>

The class d_Extent<T> provides an interface to the extent of a persistence-capable class T in the C++ binding.

d_Extent provides nearly the same interface as the d_Collection class.

```
template <class T> class d_Extent
{
public:
                        d_Extent(const d_Database* base,
                                 d_Boolean include_subclasses = d_True);
    virtual             ~d_Extent();
    unsigned long       cardinality() const;
    d_Boolean           is_empty() const;
    d_Boolean           allows_duplicates() const;
    d_Boolean           is_ordered() const;
    d_Iterator<T>       create_iterator() const;
    d_Iterator<T>       begin() const;
    d_Iterator<T>       end() const;
    d_Ref<T>            select_element(const char * OQL_pred) const;
    d_Iterator<T>       select(const char * OQL_pred) const;
    int                 query(d_Collection<d_Ref<T> > &,
                              const char* OQL_pred) const;
    d_Boolean           exists_element(const char * OQL_pred) const;
protected:
                        d_Extent(const d_Extent<T> &);
    d_Extent<T> &       operator=(const d_Extent<T> &);
};
```

The database schema definition contains a parameter for each persistent class specifying whether the ODBMS should maintain an extent for the class. This parameter can be set using the schema API or a database tool that enables specification of the schema.

The content of a d_Extent<T> is automatically maintained by the ODBMS. The d_Extent class therefore has neither insert nor remove methods. d_Extents themselves are not persistence-capable and cannot be stored in the database. This explains why d_Extent is not derived from d_Collection; since d_Collection is in turn derived from

d_Object, this would imply that extents are also persistence-capable. However, semantically d_Extent is equivalent to d_Set.

If a user wants to maintain an extent they can define a d_Set<d_Ref<T> > that is stored in the database, as in the example in Section 5.6.

The class d_Extent supports polymorphism when the constructor argument include_subclasses is a true value. If type B is a subtype of A a d_Extent for A includes all instances of A and B.

The association of a d_Extent to a type is performed by instantiating the template with the appropriate type. Every d_Extent instance must be associated with a database by passing a d_Database pointer to the constructor.

```
d_Extent<Person>    PersonExtent(database);
```

Passing the database pointer to the constructor instead of operator new (as with d_Object) allows the user to instantiate a d_Extent instance on the stack. If no extent has been defined in the database schema for the class, an exception is thrown.

Comparison operators like operator== and operator!= or the subset and superset methods of d_Set do not make sense for d_Extent, since all instances of a d_Extent for a given type have the same content.

5.3.10 Exceptions

Instances of d_Error contain state describing the cause of the error. This state is composed of a number, representing the kind of error, and optional additional information. This additional information can be appended to the error object by using operator<<. If the d_Error object is caught, more information can be appended to it if it is to be thrown again. The complete state of the object is returned as a human-readable character string by the what function.

The d_Error class is defined as follows:

```
class d_Error : public exception {
public:
    typedef d_Long    kind;
                      d_Error();
                      d_Error(const d_Error &);
                      d_Error(kind the_kind);
                      ~d_Error();
    kind              get_kind();
    void              set_kind(kind the_kind);
    const char *      what() const throw();

    d_Error &         operator<<(d_Char);
    d_Error &         operator<<(d_Short);
```

```
    d_Error &       operator<<(d_UShort);
    d_Error &       operator<<(d_Long);
    d_Error &       operator<<(d_ULong);
    d_Error &       operator<<(d_Float);
    d_Error &       operator<<(d_Double);
    d_Error &       operator<<(const char *);
    d_Error &       operator<<(const d_String &);
    d_Error &       operator<<(const d_Error &);
};
```

The null constructor initializes the kind property to d_Error_None. The class d_Error is responsible for releasing the string that is returned by the member function what.

The following constants are defined for error kinds used in this standard. Note that each of these names has a prefix of d_Error_.

Error Name	Description
None	No error has occurred.
DatabaseClassMismatch	The definition of a class in the application does not match the database definition of the class.
DatabaseClassUndefined	The database does not have the schema information about a class.
DatabaseClosed	The database is closed; objects cannot be accessed.
DatabaseOpen	The database is already open.
DateInvalid	An attempt was made to set a d_Date object to an invalid value.
ElementNotFound	An attempt was made to access an element that is not in the collection.
IteratorDifferentCollections	An iterator, passed to a member function of a collection, is not associated with the collection.
IteratorExhausted	An attempt was made to either advance an iterator once it already reached the end of a collection or move backward once the first element has been reached.
IteratorNotBackward	An attempt has been made to iterate backward with either a d_Set<T> or d_Bag<T>.
NameNotUnique	An attempt was made to associate an object with a name that is not unique in the database.

Error Name	Description
PositionOutOfRange	A position within a collection has been supplied that exceeds the range of the index (0, cardinality - 1).
QueryParameterCountInvalid	The number of arguments used to build a query with the d_OQL_Query object does not equal the number of arguments supplied in the query string.
QueryParameterTypeInvalid	Either the parameters specified in the query or the return value type does not match the types in the database.
RefInvalid	An attempt was made to dereference a d_Ref that references an object that does not exist.
RefNull	An attempt was made to dereference a null d_Ref.
TimeInvalid	An attempt has been made to set a d_Time object to an invalid value.
TimestampInvalid	An attempt has been made to set a d_Timestamp object to an invalid value.
MemberIsOfInvalidType	The second template argument of either d_Rel_Ref, d_Rel_Set, or d_Rel_List references a data member that is not of type d_Rel_Ref, d_Rel_Set, or d_Rel_List.
MemberNotFound	The second template argument of either d_Rel_Ref, d_Rel_Set, or d_Rel_List references a data member that does not exist in the referenced class.
TransactionNotOpen	A call has been made to either commit or abort without a prior call to begin.
TransactionOpen	d_Transaction::begin has been called multiple times on the same transaction object, without an intervening commit or abort.
TypeInvalid	A d_Ref<T> was initialized to reference an object that is not of type T or a subclass of T.

The following table indicates which functions throw these exceptions.

Error Name	Raised By
DatabaseClassMismatch	d_Database::lookup_object
DatabaseClassUndefined	d_Object::new
DatabaseClosed	d_Database::close
DatabaseOpen	d_Database::open
DateInvalid	Any method that alters the date value

Error Name	Raised By
IteratorExhausted	d_Iterator functions get_element, ++, --,operator *
NameNotUnique	d_Database::set_object_name
PositionOutOfRange	d_List::operator[ ], d_Varray::operator[ ]
QueryParameterCountInvalid	d_oql_execute
QueryParameterTypeInvalid	d_oql_execute
RefInvalid	d_Ref<T> functions operator–>, operator *
RefNull	d_Ref<T> functions operator–>, operator *
TimeInvalid	Any method that alters the time value
TimestampInvalid	Any method that alters the timestamp value
TransactionOpen	d_Transaction::begin
TypeInvalid	d_Ref constructor

5.4 C++ OQL

Chapter 4 outlined the Object Query Language. In this section the OQL semantics are mapped into the C++ language.

5.4.1 Query Method on Class Collection

The d_Collection class has a query member function whose signature is:

```
int query(d_Collection<T> &result, const char* predicate) const;
```

This function filters the collection using the predicate and assigns the result to the first parameter. It returns a code different from 0, if the query is not well formed. The predicate is given as a string with the syntax of the *where* clause of OQL. The predefined variable this is used inside the predicate to denote the current element of the collection to be filtered.

Example:

Given the class Student, as defined in Chapter 3, with extent referenced by Students, compute the set of students who take math courses:

```
d_Bag<d_Ref<Student> >  mathematicians;
Students–>query(mathematicians,
                    “exists s in this.takes: s.section_of.name = \“math\” ”);
```

5.4.2 d_oql_execute Function

An interface is provided to gain access to the complete functionality of OQL from a C++ program. There are several steps involved in the specification and execution of the OQL query. First, a query gets *constructed* via an object of type d_OQL_Query.

Once a query has been constructed, the query is *executed*. Once constructed, a query can be executed multiple times with different argument values.

The function to execute a query is called d_oql_execute; it is a free-standing template function, not part of any class definition:

```
template<class T> void      d_oql_execute(d_OQL_Query &query, T &result);
```

The first parameter, *query*, is a reference to a d_OQL_Query object specifying the query to execute. The second parameter, *result*, is used for returning the result of the query. The type of the query result must match the type of this second parameter, or a d_Error exception object of kind d_Error_QueryParameterTypeInvalid is thrown. Type checking of the input parameters according to their use in the query is done at runtime. Similarly, the type of the result of the query is checked. Any violation of type would cause a d_Error exception object of kind d_Error_QueryParameterTypeInvalid to be thrown. If the query returns a persistent object of type T, the function returns a d_Ref<T>. If the query returns a structured literal, the value of it is assigned to the value of the object or collection denoted by the *result* parameter.

If the result of the query is a large collection, a function d_oql_execute can be used. This function returns an iterator on the result collection instead of the collection itself. The behavior of this standalone function is exactly the same as the d_oql_execute function.

```
template<class T> void d_oql_execute (d_OQL_Query &q, d_Iterator<T> &results);
```

The << operator has been overloaded for d_OQL_Query to allow construction of the query. It concatenates the value of the right operand onto the end of the current value of the d_OQL_Query left operand. These functions return a reference to the left operand so that invocations can be cascaded.

Note that instances of d_OQL_Query contain either a partial or a complete OQL query. An ODMG implementation will contain ancillary data structures to represent a query both during its construction and once it is executed. The d_OQL_Query destructor will appropriately remove any ancillary data when the object gets deleted.

The d_OQL_Query class is defined as follows:

Definition:

```
class d_OQL_Query {
public:
                                d_OQL_Query();
                                d_OQL_Query(const char *s);
                                d_OQL_Query(const d_String &s);
                                d_OQL_Query(const d_OQL_Query &q);
                                ~d_OQL_Query();
```

```
    d_OQL_Query &       operator=(const d_OQL_Query &q);
    void                clear();

friend d_OQL_Query &    operator<<(d_OQL_Query &q, const char *s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_String &s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Char c);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Octet uc);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Short s);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_UShort us);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, int i);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, unsigned int ui);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Long l);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_ULong ul);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Float f);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, d_Double d);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Date &d);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Time &t);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Timestamp &);
friend d_OQL_Query &    operator<<(d_OQL_Query &q, const d_Interval &i);
template<class T> friend d_OQL_Query &    operator<<(d_OQL_Query &q,
                                                     const d_Ref<T> &r);
template<class T> friend d_OQL_Query & operator<<(d_OQL_Query &q,
                                                     const d_Collection<T> &c);
};
```

Strings used in the construction of a query may contain parameters signified by the form *$i*, where *i* is a number referring to the i^{th} subsequent right operand in the construction of the query; the first subsequent right operand would be referred to as $1. If any of the $i are not followed by a right operand construction argument at the point d_oql_execute is called, a d_Error exception object of kind d_Error_QueryParameterCountInvalid is thrown. This exception will also be thrown if too many parameters are used in the construction of the query. If a query argument is of the wrong type, a d_Error exception object of kind d_Error_QueryParameterTypeInvalid is thrown.

The operation clear can be called to clear the values of any query parameters that have been provided to an instance of d_OQL_Query.

Once a query has been successfully executed via d_oql_execute, the arguments associated with the *$i* parameters are cleared and new arguments must be supplied. If any exceptions are thrown, the query arguments are not implicitly cleared and must be cleared explicitly by invoking clear. The original query string containing the *$i* parameters is retained across the call to d_oql_execute.

The d_OQL_Query copy constructor and assignment operator copy all the underlying data structures associated with the query, based upon the parameters that have been passed to the query at the point the operation is performed. If the original query object had two parameters passed to it, the object that is new or assigned to should have those same two parameters initialized. After either of these operations the two d_OQL_Query objects should be equivalent and have identical behavior.

Example:

Among the math students (as computed before in Section 5.4.1 into the variable mathematicians) who are teaching assistants and earn more than x, find the set of professors that they assist. Suppose there exists a named set of teaching assistants called "TA".

```
d_Bag<d_Ref<Student> >  mathematicians; // computed as above
d_Bag<d_Ref<Professor> >  assisted_profs;
double x = 50000.00;


d_OQL_Query q1 (
      "select t.assists.taught_by from t in TA where t.salary > $1 and t in $2");
q1 << x << mathematicians;
d_oql_execute(q1, assisted_profs);
```

After the above code has been executed, it could be followed by another query, passing in different arguments.

```
d_Set<d_Ref<Student> >  historians;     // assume this gets computed similar
                                        // to mathematicians
double y = 40000.00


q1 << y << historians;
d_oql_execute(q1, assisted_profs);
```

The ODMG OQL implementation may have parsed, compiled, and optimized the original query; it can now reexecute the query with different arguments without incurring the overhead of compiling and optimizing the query.

5.5 Schema Access

This section describes an interface for accessing the schema of an ODMG database via a C++ class library. The schema information is based on the Metadata described in Chapter 2. The C++ schema definition differs in some respects from the language-independent, abstract specification of the schema in ODL. Attempts have been made to make the schema access conform to C++ programming practices in order that the use of the API be intuitive for C++ programmers. C++-specific ODL extensions have been included in the API, in addition to the abstract ODL schema, since the C++ ODL is a superset of the ODMG ODL described in Chapter 2.

The schema access API is structured as an object-oriented framework. Only the interface methods through which meta-information is manipulated is defined, rather than defining the entire class structure and details of the internal implementation. The ODBMS implementation can choose the actual physical representation of the schema database. The ODL schema lists the classes that are used to describe a schema. It also uses ODL relationships to show how instances of these classes are interrelated. To allow maximum flexibility in an implementation of this model, methods are provided to traverse the relationships among instances of these classes, rather than mapping the ODL relationships directly to C++ data members.

The specification currently contains only the *read* interface portion of the schema API. The ODMG plans to extend the specification to include a full *read/write* interface enabling dynamic modification of the schema (e.g., creation and modification of classes). The initial *read-only* interface will substantially increase the flexibility and usability of the standard. The following application domains will be able to take advantage of this interface:

- Database tool development (e.g., class and object browsers, import/export utilities, Query By Example implementations)
- CASE tools
- Schema management tools
- Distributed computing and dynamic binding, object brokers (CORBA)
- Management of extended database properties (e.g. access control and user authorization)

This concept is analogous to the system table approach used by relational database systems. However, since it is a true object mapping, it also includes schema semantics (e.g., relationships between classes and properties), making it much easier to use.

5.5.1 The ODMG Schema Access Class Hierarchy

This section defines the types in the ODMG schema access class hierarchy. Subclasses are indented and placed below their base class. The names in square brackets denote additional base classes, if the class inherits from more than one class. Types in *italic code* font are abstract classes and cannot be directly instantiated; classes in code font can be instantiated. These classes are described in more detail in later sections using both C++ and ODL syntax.

- *d_Scope*
  d_Scope instances are used to form a hierarchy of meta objects. A *d_Scope* instance contains a list of *d_Meta_Object* instances, which are defined in the scope; operations to manage the list (e.g. finding a *d_Meta_Object* by its name) are provided. All meta objects are defined in a scope.

- *d_Meta_Object*
  Instances of *d_Meta_Object* are used to describe elements of the schema stored in the dictionary.
  - d_Module [*d_Scope*]
    d_Module instances manage domains in a dictionary. They are used to group and order *d_Meta_Object* instances, such as type/class descriptions, constant descriptions, subschemas (expressed as d_Module objects), etc.
  - d_Type
    d_Type is an abstract base class for all type descriptions.
    - d_Class [*d_Scope*]
      A d_Class instance is used to describe an application-defined class whose d_Attribute and d_Relationship instances represent the concrete state of an instance of that class. The state is stored in the database. All persistent-capable classes are described by a d_Class instance.
    - d_Ref_Type
      d_Ref_Type instances are used to describe types that are references to other objects. References can be pointers, references (like d_Ref<T>), or other language-specific references. The referenced object or literal can be shared by more than one reference, i.e., multiple references can reference the same object.
    - d_Collection_Type
      A d_Collection_Type describes a type whose instances group a set of elements in a collection. The collection elements must be of the same (base) type.
      - d_Keyed_Collection_Type
        A d_Keyed_Collection_Type describes a collection that can be accessed via keys.
    - d_Primitive_Type
      A d_Primitive_Type represents all built-in types, e.g., int (16, 32 bit), float, etc., as well as pre-defined ODMG literals such as d_String.
      - d_Enumeration_Type [*d_Scope*]
        d_Enumeration_Type describes a type whose domain is a list of identifiers.
    - d_Structure_Type [*d_Scope*]
      d_Structure_Type instances describe application-defined member values. The members are described by d_Attribute instances and

represent the state of the structure. Structures do not have object identity.

- d_Alias_Type
  A d_Alias_Type describes a type that is equivalent to another *d_Type*, but has another name.

- d_Property
  d_Property is an abstract base class for all *d_Meta_Object* instances that describe the state (abstract or concrete) of application-defined types.
  - d_Relationship
    Instances of d_Relationship describe relationships between persistent objects; in C++ these are expressed as d_Rel_Ref<T, MT>, d_Rel_Set<T, MT>, and d_Rel_List<T, MT> data members of a class.
  - d_Attribute
    A d_Attribute instance describes the concrete state of an object or structure.
- d_Operation [*d_Scope*]
  d_Operation instances describe methods, including their return type, identifier, signature, and list of exceptions.
- d_Exception
  d_Exception instances describe exceptions that are raised by operations represented by instances of d_Operation.
- d_Parameter
  A d_Parameter describes a parameter of an operation. A parameter has a name, a type, and a mode (in, out, inout).
- d_Constant
  A d_Constant describes a value that has a name and a type. The value may not be changed.

- d_Inheritance
  d_Inheritance is used to describe the bidirectional relationship between a base class and a subclass, as well as the type of inheritance used.

5.5.2 Schema Access Interface

The following interfaces describe the external C++ interface of an ODMG 2.0 schema repository. The interface is defined in terms of C++ classes with public methods, without exposing or suggesting any particular implementation.

In the following specification, some hints are provided as to how the ODL interface repository described in Chapter 2 has been mapped into the C++ interface.

All objects in the repository are subclasses of *d_Meta_Object* or *d_Scope*.

Interfaces that return a list of objects are expressed using iterator types, traversal of proposed relationships is expressed via methods that return iterators, access to attributes is provided using accessor functions.

An implementation of this interface is not required to use a particular implementation of the iterator type. We follow a design principle used in STL. Classes that have 1-to-n relationships to other classes define only a type to iterate over this relationship. A conformant implementation of the schema repository can implement these types by means of the d_Iterator type (but is not required to).

The iterator protocol must support at least the following methods. In subsequent sections this set of methods for iterator type *IterType* is referred to as a "constant forward iterator" over type T.

```
                    IterType();
                    IterType(const IterType &);
IterType &          operator=(const IterType &);
int                 operator==(const IterType &) const;
int                 operator!=(const IterType &) const;
IterType &          operator++();
IterType            operator++(int);
const  T &          operator*() const;
```

The operator++ advances the iterator to the next element, and operator* retrieves the element. The iterator is guaranteed to always return the elements in the same order. The iterator in class d_Scope that iterates over instances of type d_Meta_Object would be of type d_Scope::meta_object_iterator.

The following names are used for embedded types that provide constant forward iteration.

Iterator Type Name	Element Type
alias_type_iterator	d_Alias_Type
attribute_iterator	d_Attribute
collection_type_iterator	d_Collection_Type
constant_iterator	d_Constant
exception_iterator	d_Exception
inheritance_iterator	d_Inheritance
keyed_collection_type_iterator	d_Keyed_Collection_Type
operation_iterator	d_Operation
parameter_iterator	d_Parameter
property_iterator	d_Property

Iterator Type Name	Element Type
ref_type_iterator	d_Ref_Type
relationship_iterator	d_Relationship
type_iterator	d_Type

Each class that iterates over elements of one of these types has an iterator type defined within the class with the corresponding name. This is denoted in the class interface with a line similar to the following:

```
typedef property_iterator; // this implies an iterator type with this name is defined
```

Properties and classes have access specifiers in C++. Several of the metaclass objects make use of the following enumeration:

```
typedef enum { d_PUBLIC, d_PROTECTED, d_PRIVATE } d_Access_Kind;
```

5.5.2.1 d_Scope

d_Scope instances are used to form a hierarchy of meta objects. A *d_Scope* contains a list of *d_Meta_Object* instances that are defined in the scope, as well as operations to manage the list. The method resolve is used to find a *d_Meta_Object* by name. All instances of *d_Meta_Object*, except d_Module, have exactly one *d_Scope* object. This represents a "defined in" relationship. The type *d_Scope*::meta_object_iterator defines a protocol to traverse this relationship in the other direction.

```
class d_Scope {
public:
    const d_Meta_Object &    resolve(const char *name) const;
    typedef                  meta_object_iterator;
    meta_object_iterator     defines_begin() const;
    meta_object_iterator     defines_end() const;
};
```

5.5.2.2 d_Meta_Object

Class *d_Meta_Object* has a name, an id and a comment attribute. Some instances of *d_Meta_Object* are themselves scopes (instances of *d_Scope*); that is, they define a namespace in which other *d_Meta_Object* instances can be identified (resolved) by name. They form a defines/defined_in relationship with other *d_Meta_Object* instances and are their defining scopes. The scope of a *d_Meta_Object* is obtained by the method defined_in.

```
class d_Meta_Object {
public:
    const char *             name() const;
    const char *             comment() const;
    const d_Scope &          defined_in() const;
};
```

5.5.2.3 d_Module

A d_Module manages domains in a dictionary. They are used to group and order *d_Meta_Object* instances such as type/class descriptions, constant descriptions, subschemas (expressed as d_Module objects), etc. A d_Module is also a *d_Scope* that provides client repository services. A module is the uppermost meta object in a naming hierarchy. The class d_Module provides methods to iterate over the various meta objects that can be defined in a module. It is an entry point for accessing instances of *d_Type*, d_Constant, and d_Operation. The *d_Type* objects returned by type_iterator can be asked for the set of d_Operations, *d_Properties*, etc. that describe operations and properties of the module. It is then possible to navigate further down in the hierarchy. For example, from d_Operation, the set of d_Parameter instances can be reached, and so on.

```
class d_Module : public d_Meta_Object, public d_Scope {
public:
    static const d_Module &    top_level(const d_Database &);

    typedef                    type_iterator;
    type_iterator              defines_types_begin() const;
    type_iterator              defines_types_end() const;

    typedef                    constant_iterator;
    constant_iterator          defines_constant_begin() const;
    constant_iterator          defines_constant_end() const;

    typedef                    operation_iterator;
    operation_iterator         defines_operation_begin() const;
    operation_iterator         defines_operation_end() const;
};
```

5.5.2.4 d_Type

d_Type meta objects are used to represent information about data types. They participate in a number of relationships with the other *d_Meta_Objects*. These relationships allow types to be easily administered within the repository and help to ensure the referential integrity of the repository as a whole.

Class *d_Type* is defined as follows:

```
class d_Type : public d_Meta_Object {
public:
    typedef                    alias_type_iterator;
    alias_type_iterator        used_in_alias_type_begin() const;
    alias_type_iterator        used_in_alias_type_end() const;
```

```
    typedef                         collection_type_iterator;
    collection_type_iterator        used_in_collection_type_begin() const;
    collection_type_iterator        used_in_collection_type_end() const;

    typedef                         keyed_collection_type_iterator;
    keyed_collection_type_iterator  used_in_keyed_collection_type_begin() const;
    keyed_collection_type_iterator  used_in_keyed_collection_type_end() const;

    typedef                         ref_type_iterator;
    ref_type_iterator               used_in_ref_type_begin() const;
    ref_type_iterator               used_in_ref_type_end() const;

    typedef                         property_iterator;
    property_iterator               used_in_property_begin() const;
    property_iterator               used_in_property_end() const;

    typedef                         operation_iterator;
    operation_iterator              used_in_operation_begin() const;
    operation_iterator              used_in_operation_end() const;

    typedef                         exception_iterator;
    exception_iterator              used_in_exception_begin() const;
    exception_iterator              used_in_exception_end() const;

    typedef                         parameter_iterator;
    parameter_iterator              used_in_parameter_begin() const;
    parameter_iterator              used_in_parameter_end() const;

    typedef                         constant_iterator;
    constant_iterator               used_in_constant_begin() const;
    constant_iterator               used_in_constant_end() const;
};
```

5.5.2.5 d_Class

A d_Class object describes an application-defined type whose attributes and relationships form the concrete state of an object of that type. The state is stored in the database. All persistent-capable classes are described by a d_Class instance.

d_Class objects are linked in a multiple inheritance graph by two relationships, inherits and derives. The relationship between two d_Class objects is formed by means of one

connecting d_Inheritance object. A d_Class object also indicates whether the database maintains an extent for the class.

A class defines methods, data and relationship members, constants, and types; that is, the class is their defining scope.

Methods, data and relationship members, constants, and types are modeled by a list of related objects of type d_Operation, d_Attribute, d_Relationship, d_Constant, and *d_Type*. These descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The methods resolve_operation, resolve_attribute, or resolve_constant can be used as shortcuts.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns descriptions for methods, data members, relationship members, constants, and types. Methods, data members, relationship members, constants, and types are also accessible via special iterators. The following functions provide iterators that return their elements in their declaration order:

- base_class_list_begin
- defines_attribute_begin
- defines_operation_begin
- defines_constant_begin
- defines_relationship_begin
- defines_type_begin

The class d_Class is defined as follows:

```
class d_Class : public d_Type, public d_Scope {
public:
    typedef                 inheritance_iterator;
    inheritance_iterator    sub_class_list_begin() const;
    inheritance_iterator    sub_class_list_end() const;
    inheritance_iterator    base_class_list_begin() const;
    inheritance_iterator    base_class_list_end() const;

    d_Boolean               persistent_capable() const; // derived from d_Object?

    // these methods are used to return the characteristics of the class
    typedef                 operation_iterator;
    operation_iterator      defines_operation_begin() const;
    operation_iterator      defines_operation_end() const;
    const d_Operation &     resolve_operation(const char *name) const;

    typedef                 attribute_iterator;
    attribute_iterator      defines_attribute_begin() const;
```

```
    attribute_iterator      defines_attribute_end() const;
    const d_Attribute &     resolve_attribute(const char *name) const;

    typedef                 relationship_iterator;
    relationship_iterator   defines_relationship_begin() const;
    relationship_iterator   defines_relationship_end() const;
    const d_Relationship& resolve_relationship(const char *name) const;

    typedef                 constant_iterator;
    constant_iterator       defines_constant_begin() const;
    constant_iterator       defines_constant_end() const;
    const d_Constant &      resolve_constant(const char *name) const;

    typedef type_iterator;
    type_iterator           defines_type_begin() const;
    type_iterator           defines_type_end() const;
    const d_Type &          resolve_type(const char *name) const;

    d_Boolean               has_extent() const;
};
```

5.5.2.6 d_Ref_Type

d_Ref_Type instances are used to describe types that are references to other types. References can be pointers, references (like d_Ref<T>), or other language-specific references. The referenced object or literal can be shared by more than one reference.

```
class d_Ref_Type : public d_Type {
public:
    typedef enum { REF, POINTER } d_Ref_Kind;
    d_Ref_Kind              ref_kind() const;
    const d_Type &          referenced_type() const;
};
```

5.5.2.7 d_Collection_Type

A d_Collection_Type describes a type that aggregates a variable number of elements of a single type and provides ordering, accessing, and comparison functionality.

```
class d_Collection_Type : public d_Type {
public:
    typedef enum {  LIST,
                    ARRAY,
                    BAG,
```

```
                    SET,
                    DICTIONARY,
                    STL_LIST,
                    STL_SET,
                    STL_MULTISET,
                    STL_VECTOR,
                    STL_MAP,
                    STL_MULTIMAP } d_Kind;

    d_Kind          kind() const;
    const d_Type &  element_type() const;
};
```

5.5.2.8 d_Keyed_Collection_Type

A d_Keyed_Collection_Type describes a collection type whose element can be accessed via keys. Examples are dictionaries and maps.

```
class d_Keyed_Collection_Type : public d_Collection_Type {
public:
    const d_Type &  key_type() const;
    const d_Type &  value_type() const;
};
```

5.5.2.9 d_Primitive_Type

d_Primitive_Type objects represent built-in types. These types are atomic; they are not composed of other types.

```
class d_Primitive_Type : public d_Type {
public:
    typedef enum {
            CHAR,
            SHORT,
            LONG,
            DOUBLE,
            FLOAT,
            USHORT,
            ULONG,
            OCTET,
            BOOLEAN,
            ENUMERATION } d_TypeId;

    d_TypeId        type_id() const;
};
```

5.5.2.10 d_Enumeration_Type

A d_Enumeration_Type describes a type whose domain is a list of identifiers.

An enumeration defines a scope for its identifiers. These identifiers are modeled by a list of related d_Constant objects. The d_Constant objects accessed via the iterator returned by defines_constant_begin are returned in their declaration order.

Constant descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_constant can also be used as a shortcut. The inherited iterator *d_Meta_Object*::meta_object_iterator returns enumeration member descriptions of type d_Constant.

The name of the constant descriptions is equivalent to the domain of the enumeration identifiers. All constants of an enumeration must be of the same discrete type. The enumeration identifiers are associated with values of this discrete type. For instance, an enumeration “days_of_week” has the domain “Monday,” “Tuesday,” “Wednesday,” and so on. The enumeration description refers to a list of seven constant descriptions. The names of these descriptions are named “Monday,” “Tuesday,” “Wednesday,” and so on. All these descriptions reference the same type description, here an object of type d_Primitive_Type with the name “int”. The values of the constants are integers, e.g., 1, 2, 3, and so on up to 7, and can be obtained from the constant description.

```
class d_Enumeration_Type : public d_Primitive_Type, public d_Scope{
public:
    typedef constant_iterator;
    constant_iterator       defines_constant_begin() const;
    constant_iterator       defines_constant_end() const;
    const d_Constant &      resolve_constant(const char *name) const;
};
```

5.5.2.11 d_Structure_Type

d_Structure_Type describes application-defined aggregated values. The members represent the state of the structure. Structures have no identity.

A structure defines a scope for its members. These members are modeled using a list of related d_Attribute objects. The member descriptions can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_attribute can be used as a shortcut.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns member descriptions of type d_Attribute. Structure members are also accessible via the iterator returned by defines_attribute_begin, which returns them during iteration in the order they are declared in the structure.

```
class d_Structure_Type : public d_Type, public d_Scope {
public:
    typedef                 attribute_iterator;
    attribute_iterator      defines_attribute_begin() const;
    attribute_iterator      defines_attribute_end() const;
    const d_Attribute &     resolve_attribute(const char *name) const;
};
```

5.5.2.12 d_Alias_Type

A d_Alias_Type describes a type that is equivalent to another type, but has another name. The description of the related type is returned by the method alias_type.

The defining scope of a type alias is either a module or a class; the inherited method *d_Meta_Object*::defined_in returns an object of class d_Class or d_Module.

```
class d_Alias_Type : public d_Type {
public:
    const d_Type &          alias_type() const;
};
```

5.5.2.13 d_Property

d_Property is an abstract base class for d_Attribute and d_Relationship. Properties have a name and a type. The name is returned by the inherited method *d_Meta_Object*::name. The type description can be obtained using the method type_of.

Properties are defined in the scope of exactly one structure or class. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Structure_Type or d_Class, respectively.

```
class d_Property : public d_Meta_Object {
public:
    const d_Type &          type_of() const;
    d_Access_Kind           access_kind() const;
};
```

5.5.2.14 d_Attribute

d_Attribute describes a member of an object or a literal. An attribute has a name and a type. The name is returned by the inherited method *d_Meta_Object*::name. The type description of an attribute can be obtained using the inherited method d_Property::type_of.

Attributes may be read-only, in which case their values cannot be changed. This is described in the meta object by the method is_read_only. If an attribute is a static data member of a class, the method is_static returns d_True.

Attributes are defined in the scope of exactly one class or structure. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Class or d_Structure_Type, respectively.

```
class d_Attribute : public d_Property {
public:
    d_Boolean       is_read_only() const;
    d_Boolean       is_static() const;
    unsigned long   dimension() const;
};
```

5.5.2.15 d_Relationship

d_Relationships model bilateral object references between participating objects. In practice, two relationship meta objects are required to represent each traversal direction of the relationship. Operations are defined implicitly to form and drop the relationship, as well as accessor operations for traversing the relationship. The inherited *d_Type* expresses the cardinality. It may be either a d_Rel_Ref, d_Rel_Set, or d_Rel_List; the method rel_kind returns a d_Rel_Kind enumeration indicating the type.

The defining scope of a relationship is a class. The inherited method *d_Meta_Object*::defined_in returns a d_Class object. The method defined_in_class can be used as a shortcut.

```
class d_Relationship : public d_Property {
public:
    typedef enum { REL_REF, REL_SET, REL_LIST } d_Rel_Kind;
    d_Rel_Kind                  rel_kind() const;
    const d_Relationship &      inverse() const;
    const d_Class &             defined_in_class() const;
};
```

5.5.2.16 d_Operation

d_Operation describes the behavior supported by application objects. Operations have a name, a return type, and a signature (list of parameters), which is modeled by the inherited method *d_Meta_Object*::name, a *d_Type* object returned by result_type, and a list of d_Parameter objects (accessible via an iterator). The d_Parameter objects are returned during iteration in the order that they are declared in the operation. Operations may raise exceptions. The list of possible exceptions is described by a list of d_Exception objects (accessible via an iterator).

Operations may have an access specifier. This is described by the method access_kind inherited from d_Property.

An operation defines a scope for its parameters. They can be accessed by name using the inherited method *d_Scope*::resolve. The method resolve_parameter can be used as a shortcut.

The inherited iterator *d_Meta_Object*::meta_object_iterator returns a parameter description of type d_Parameter. Parameters are also accessible via a special parameter_iterator.

The defining scope for an operation is either a class or a module.

The inherited method *d_Meta_Object*::defined_in returns a d_Class object.

```
class d_Operation : public d_Meta_Object, public d_Scope {
public:
    const d_Type &          result_type () const;
    d_Boolean               is_static() const;

    typedef                 parameter_iterator;
    parameter_iterator      defines_parameter_begin() const;
    parameter_iterator      defines_parameter_end() const;
    const d_Parameter &     resolve_parameter(const char *name) const;

    typedef                 exception_iterator;
    exception_iterator      raises_exception_begin() const;
    exception_iterator      raises_exception_end() const;
    d_Access_Kind           access_kind() const;
};
```

If the operation is a static member function of a class, the method is_static returns d_True.

5.5.2.17 d_Exception

Operations may raise exceptions. A d_Exception describes such an exception. An exception has a name, which can be accessed using the inherited method *d_Meta_Object*::name, and a type whose description can be obtained using the method exception_type.

A single exception can be raised in more than one operation. The list of operation descriptions can be accessed via an iterator.

The defining scope of an exception is a module. The inherited method *d_Meta_Object*::defined_in returns a d_Module object. The method defined_in_module can be used as a shortcut.

```
class d_Exception : public d_Meta_Object {
```

```
public:
    const d_Type &          exception_type() const;
    typedef                 operation_iterator;
    operation_iterator      raised_in_operation_begin() const;
    operation_iterator      raised_in_operation_end() const;
    const d_Module &        defined_in_module() const;
};
```

5.5.2.18 d_Parameter

d_Parameter describes a parameter of an operation. Parameters have a name, a type, and a mode (in, out and inout). The name is returned by the inherited method *d_Meta_Object*::name.

The type description can be obtained by the method parameter_type. The mode is returned by the method mode.

Parameters are defined in the scope of exactly one operation, and the inherited method *d_Meta_Object*::defined_in returns an object of class d_Operation. The method defined_in_operation can be used as a shortcut.

```
class d_Parameter : public d_Meta_Object {
public:
    typedef enum { IN, OUT, INOUT } d_Mode;
    d_Mode                  mode() const;
    const d_Type &          parameter_type() const;
    const d_Operation &     defined_in_operation() const;
};
```

5.5.2.19 d_Constant

Constants provide a mechanism for statically associating values with names in the repository. Constants are used by enumerations to form domains. In this case, the name of a d_Constant is used as an identifier for an enumeration value. Its name is returned by the inherited method *d_Meta_Object*::name.

Constants are defined in the scope of exactly one module or class. The inherited method *d_Meta_Object*::defined_in returns an object of class d_Module or d_Class, respectively.

```
class d_Constant : public d_Meta_Object {
public:
    const d_Type &          constant_type() const;
    void *                  constant_value() const;
};
```

5.5.2.20 d_Inheritance

d_Inheritance is used to describe the bidirectional relationship between a base class and a subclass, as well as the type of inheritance used. An object of type d_Inheritance connects two objects of type d_Class.

Depending on the programming language, inheritance relationships can have properties. The schema objects that describe inheritance relationships are augmented with information to reproduce the language-specific extensions.

```
class d_Inheritance {
public:
    const d_Class &     derives_from() const;
    const d_Class &     inherits_to() const;

    d_Access_Kind       access_kind() const;
    d_Boolean           is_virtual() const;
};
```

5.6 Example

This section gives a complete example of a small C++ application. This application manages people records. A Person may be entered into the database. Then special events can be recorded: marriage, the birth of children, moving to a new address.

The application comprises two transactions: the first one populates the database, while the second consults and updates it.

The next section defines the schema of the database, as C++ ODL classes. The C++ program is given in the subsequent section.

5.6.1 Schema Definition

For the explanation of the semantics of this example, see Section 3.2.3. Here is the C++ ODL syntax:

```
// Schema Definition in C++ ODL
class City;             // forward declaration
struct Address {
    d_UShort            number;
    d_String            street;
    d_Ref<City>         city;
                        Address();
                        Address(d_UShort, const char*, const d_Ref<City> &);
};
```

```
extern const char _spouse [ ],  _parents [ ],  _children [ ];

class Person : public d_Object {
public:
// Attributes (all public, for this example)
     d_String                    name;
     Address                     address;
// Relationships
     d_Rel_Ref<Person, _spouse>  spouse;
     d_Rel_List<Person, _parents>  children;
     d_Rel_List<Person, _children> parents;
// Operations
                                 Person(const char * pname);
     void                        birth(const d_Ref<Person> &child); // a child is born
     void                        marriage(const d_Ref<Person> &to_whom);
     d_Ref<d_Set<d_Ref<Person> > >ancestors() const; // returns ancestors
     void                        move(const Address &); // move to a new address
// Extent
static  d_Ref<d_Set<d_Ref<Person> > >     people; // a reference to class extent[1]
static  const char * const                extent_name;
};
class City : public d_Object {
public:
// Attributes
     d_ULong                          city_code;
     d_String                         name;
     d_Ref<d_Set<d_Ref<Person> > >    population;  // the people living in this city
// Operations
                                      City(int, const char*);
// Extension
static   d_Ref<d_Set<d_Ref<City> > >  cities;      // a reference to the class extent
static  const char * const            extent_name;
};
```

1. This (transient) static variable will be initialized at transaction begin time (see the application).

5.6.2 Schema Implementation

We now define the code of the operations declared in the schema. This is written in plain C++. We assume the C++ ODL preprocessor has generated a file, “schema.hxx”, which contains the standard C++ definitions equivalent to the C++ ODL classes.

```
// Classes Implementation in C++
#include "schema.hxx"

const char _spouse [ ] = "spouse";
const char _parents [ ] = "parents";
const char _children [ ] = "children";

// Address structure:

Address::Address(d_UShort pnum, const char* pstreet,
                         const d_Ref<City> &pcity)
  : number(pnumber),
    street(pstreet),
    city(pcity)
{ }

Address::Address()
  : number(0),
    street(0),
    city(0)
{ }
// Person Class:
const char * const Person::extent_name = "people";
Person::Person(const char * pname)
  : name(pname)
{
    people->insert_element(this);  // Put this person in the extension
}
void Person::birth(const d_Ref<Person> &child)
{                                        // Adds a new child to the children list
    children.insert_element_last(child);
    if(spouse)
        spouse->children.insert_element_last(child);
}
void Person::marriage(const d_Ref<Person> &to_whom)
{                                    // Initializes the spouse relationship
```

```
    spouse = with;                  // with–>spouse is automatically set to this person
}
d_Ref<d_Set<d_Ref<Person> > >  Person::ancestors()
{                                   // Constructs the set of all ancestors of this person
    d_Ref<d_Set<d_Ref<Person> > >  the_ancestors =
                                        new d_Set<d_Ref<Person> >;
    int i;
    for( i = 0; i < 2; i++)
        if( parents[i] ) {
            // The ancestors = parents union ancestors(parents)
            the_ancestors–>insert_element(parents[i]);
            d_Ref<d_Set<d_Ref<Person> > >
                grand_parents= parents[i]–>ancestors();
            the_ancestors–>union_with(*grand_parents);
            grand_parents.delete_object();
        }
    return the_ancestors;
}
void Person::move(const Address &new_address)
{                                   // Updates the address attribute of this person
    if(address.city)
        address.city–>population–>remove_element(this);
    new_address.city–>population–>insert_element(this);
    mark_modified();[2]
    address = new_address;
}
// City class:

const char * const City::extent_name = "cities";

City::City(d_ULong code, const char * cname)
  : city_code(code),
    name(cname)
{
    cities–>insert_element(this);           // Put this city into the extension
}
```

2. Do not forget it! Notice that it is necessary only in the case where an attribute of the object is modified. When a relationship is updated, the object is automatically marked modified.

5.6.3 An Application

We now have the whole schema well defined and implemented. We are able to populate the database and play with it. In the following application, the transaction Load builds some objects into the database. Then the transaction Consult reads it, prints some reports from it, and makes updates. Each transaction is implemented inside a C++ function.

The database is opened by the main program, which then starts the transactions.

```
#include <iostream.h>
#include "schema.hxx"

static d_Database dbobj;
static d_Database * database = &dbobj;

void Load()
{                           // Transaction that populates the database
    d_Transaction load;
    load.begin();
    // Create both persons and cities extensions, and name them.

    Person::people = new(database) d_Set<d_Ref<Person> >;
    City::cities = new(database) d_Set<d_Ref<City> >;

    database->set_object_name(Person::people, Person::extent_name);
    database->set_object_name(City::cities, City::extent_name);

    // Construct 3 persistent objects from class Person.

    d_Ref<Person> God, Adam, Eve;

    God  = new(database, "Person") Person("God");
    Adam = new(database, "Person") Person("Adam");
    Eve  = new(database, "Person") Person("Eve");

// Construct an Address structure, Paradise, as (7 Apple Street, Garden),
// and set the address attributes of Adam and Eve.

    Address Paradise(7, "Apple", new(database, "City") City(0, "Garden"));

    Adam->move(Paradise);
    Eve->move(Paradise);
```

```
// Define the family relationships
        God–>birth(Adam);
        Adam–>marriage(Eve);
        Adam–>birth(new(database, "Person") Person("Cain"));
        Adam–>birth(new(database, "Person") Person("Abel"));

        load.commit();      // Commit transaction, putting objects into the database
}

static void print_persons(const d_Collection<d_Ref<Person> >& s)
{                                   // A service function to print a collection of persons
        d_Ref<Person> p;
        d_Iterator<d_Ref<Person> > it = s.create_iterator();
        while( it.next(p) ) {
                cout << "--- " << p–>name << " lives in ";
                if (p–>address.city)
                        cout << p->address.city–>name;
                else
                        cout << "Unknown";
                cout << endl;
        }
}

void Consult()
{                               // Transaction that consults and updates the database
        d_Transaction                   consult;
        d_List<d_Ref<Person> >  list;
        d_Bag<d_Ref<Person>>    bag;
        consult.begin();
        // Static references to objects or collections must be recomputed
        // after a commit
        Person::people = database–>lookup_object(Person::extent_name);
        City::cities = database–>lookup_object(City::extent_name);
        // Now begin the transaction
        cout << "All the people ....:" << endl;
        print_persons(*Person::people);
        cout << "All the people sorted by name ....:" << endl;
        d_oql_execute("select p from people order by name", list);
        print_persons(list);
        cout << "People having 2 children and living in Paradise ...:" << endl;
```

```
    d_oql_execute(list, "select p from p in people\
            where   p.address.city.name = \"Garden\"\
                    and count(p.children) = 2", bag);
    print_persons(bag);
    // Adam and Eve are moving ...
    Address Earth(13, "Macadam", new(database, "City") City(1, "St-Croix"));
    d_Ref<Person> Adam;
    d_oql_execute("element(select p from p in people\
            where p.name = \"Adam\")", Adam);
    Adam->move(Earth);
    Adam->spouse->move(Earth);
    cout << "Cain's ancestors ...:" << endl;
    d_Ref<Person> Cain = Adam->children.retrieve_element_at(0);
    print_persons(*(Cain->ancestors()));
    consult.commit();
}

main()
 {
    database->open("family");
    Load();
    Consult();
    database->close();
}
```