C++ Object Persistence with ODB

Copyright © 2009-2025 Code Synthesis.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, version 1.3; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts.

Revision 2.5, January 2025

This revision of the manual describes ODB 2.5.0 and is available in the following formats: XHTML, PDF, and PostScript.

Table of Contents

Preface 1
About This Document 1
More Information 2
PART I OBJECT-RELATIONAL MAPPING 3
1 Introduction 4
1.1 Architecture and Workflow 5
1.2 Benefits 8
1.3 Supported C++ Standards 9
2 Hello World Example 10
2.1 Declaring Persistent Classes 10
2.2 Generating Database Support Code 13
2.3 Compiling and Running 14
2.4 Making Objects Persistent 15
2.5 Querying the Database for Objects 19
2.6 Updating Persistent Objects 21
2.7 Defining and Using Views 23
2.8 Deleting Persistent Objects 25
2.9 Changing Persistent Classes 26
2.10 Working with Multiple Databases 28
2.11 Summary 30
3 Working with Persistent Objects 31
3.1 Concepts and Terminology 31
3.2 Declaring Persistent Objects and Values 33
3.3 Object and View Pointers 36
3.4 Database 38
3.5 Transactions 41
3.6 Connections 46
3.7 Error Handling and Recovery 48
3.8 Making Objects Persistent 49
3.9 Loading Persistent Objects 51
3.10 Updating Persistent Objects 52
3.11 Deleting Persistent Objects 54
3.12 Executing Native SQL Statements 57
3.13 Tracing SQL Statement Execution 57
3.14 ODB Exceptions 60
4 Querying the Database 67
4.1 ODB Query Language 68
4.2 Parameter Binding 70
4.3 Executing a Query 71
4.4 Query Result 74

4.5 Prepared Queries 78
5 Containers 87
5.1 Ordered Containers 88
5.2 Set and Multiset Containers 90
5.3 Map and Multimap Containers 91
5.4 Change-Tracking Containers 92
5.4.1 Change-Tracking `vector` 95
5.5 Using Custom Containers 98
6 Relationships 99
6.1 Unidirectional Relationships 102
6.1.1 To-One Relationships 103
6.1.2 To-Many Relationships 103
6.2 Bidirectional Relationships 105
6.2.1 One-to-One Relationships 108
6.2.2 One-to-Many Relationships 109
6.2.3 Many-to-Many Relationships 110
6.3 Circular Relationships 111
6.4 Lazy Pointers 114
6.5 Using Custom Smart Pointers 119
7 Value Types 121
7.1 Simple Value Types 121
7.2 Composite Value Types 121
7.2.1 Composite Object Ids 125
7.2.2 Composite Value Column and Table Names 125
7.3 Pointers and `NULL` Value Semantics 128
8 Inheritance 133
8.1 Reuse Inheritance 135
8.2 Polymorphism Inheritance 137
8.2.1 Performance and Limitations 142
8.3 Mixed Inheritance 145
9 Sections 146
9.1 Sections and Inheritance 153
9.2 Sections and Optimistic Concurrency 155
9.3 Sections and Lazy Pointers 157
9.4 Sections and Change-Tracking Containers 157
10 Views 159
10.1 Object Views 161
10.2 Object Loading Views 167
10.3 Table Views 174
10.4 Mixed Views 177
10.5 View Query Conditions 178
10.6 Native Views 180
10.7 Other View Features and Limitations 182

11 Session 184
11.1 Object Cache 187
11.2 Custom Sessions 188
12 Optimistic Concurrency 191
13 Database Schema Evolution 197
13.1 Object Model Version and Changelog 198
13.2 Schema Migration 206
13.3 Data Migration 214
13.3.1 Immediate Data Migration 215
13.3.2 Gradual Data Migration 222
13.4 Soft Object Model Changes 224
13.4.1 Reuse Inheritance Changes 230
13.4.2 Polymorphism Inheritance Changes 232
14 ODB Pragma Language 233
14.1 Object Type Pragmas 236
14.1.1 `table` 236
14.1.2 `pointer` 237
14.1.3 `abstract` 238
14.1.4 `readonly` 239
14.1.5 `optimistic` 239
14.1.6 `no_id` 240
14.1.7 `callback` 240
14.1.8 `schema` 242
14.1.9 `polymorphic` 246
14.1.10 `session` 246
14.1.11 `definition` 247
14.1.12 `transient` 247
14.1.13 `sectionable` 247
14.1.14 `deleted` 247
14.1.15 `bulk` 247
14.1.16 `options` 248
14.2 View Type Pragmas 248
14.2.1 `object` 249
14.2.2 `table` 249
14.2.3 `query` 249
14.2.4 `pointer` 249
14.2.5 `callback` 249
14.2.6 `definition` 249
14.2.7 `transient` 250
14.3 Value Type Pragmas 250
14.3.1 `type` 252
14.3.2 `id_type` 252
14.3.3 `null/not_null` 253

14.3.4 default 254
14.3.5 options 255
14.3.6 readonly 255
14.3.7 definition 255
14.3.8 transient 257
14.3.9 unordered 257
14.3.10 index_type 257
14.3.11 key_type 257
14.3.12 value_type 257
14.3.13 value_null/value_not_null 258
14.3.14 id_options 258
14.3.15 index_options 258
14.3.16 key_options 259
14.3.17 value_options 259
14.3.18 id_column 259
14.3.19 index_column 259
14.3.20 key_column 260
14.3.21 value_column 260
14.4 Data Member Pragmas 260
14.4.1 id 262
14.4.2 auto 262
14.4.3 type 263
14.4.4 id_type 263
14.4.5 get/set/access 264
14.4.6 null/not_null 268
14.4.7 default 269
14.4.8 options 271
14.4.9 column (object, composite value) 272
14.4.10 column (view) 272
14.4.11 transient 272
14.4.12 readonly 273
14.4.13 virtual 274
14.4.14 inverse 279
14.4.15 on_delete 280
14.4.16 version 282
14.4.17 index 282
14.4.18 unique 283
14.4.19 unordered 283
14.4.20 table 284
14.4.21 load/update 284
14.4.22 section 285
14.4.23 added 285
14.4.24 deleted 285

14.4.25 index_type . . . 285
14.4.26 key_type . . . 285
14.4.27 value_type . . . 286
14.4.28 value_null/value_not_null . . . 286
14.4.29 id_options . . . 287
14.4.30 index_options . . . 287
14.4.31 key_options . . . 288
14.4.32 value_options . . . 288
14.4.33 id_column . . . 288
14.4.34 index_column . . . 289
14.4.35 key_column . . . 289
14.4.36 value_column . . . 289
14.4.37 points_to . . . 290
14.5 Namespace Pragmas . . . 290
14.5.1 pointer . . . 291
14.5.2 table . . . 292
14.5.3 schema . . . 293
14.5.4 session . . . 293
14.6 Object Model Pragmas . . . 294
14.6.1 version . . . 294
14.7 Index Definition Pragmas . . . 294
14.8 Database Type Mapping Pragmas . . . 298
14.8.1 C++ Type Mapping Pragmas . . . 298
14.8.2 Database Type Mapping Pragmas . . . 299
14.9 C++ Compiler Warnings . . . 303
14.9.1 GNU C++ . . . 304
14.9.2 Visual C++ . . . 304
14.9.3 Sun C++ . . . 305
14.9.4 IBM XL C++ . . . 305
14.9.5 HP aC++ . . . 305
14.9.6 Clang . . . 305
15 Advanced Techniques and Mechanisms . . . 307
15.1 Transaction Callbacks . . . 307
15.2 Persistent Class Template Instantiations . . . 310
15.3 Bulk Database Operations . . . 311
PART II DATABASE SYSTEMS . . . 319
16 Multi-Database Support . . . 320
16.1 Static Multi-Database Support . . . 323
16.2 Dynamic Multi-Database Support . . . 326
16.2.2 Dynamic Loading of Database Support Code . . . 329
17 MySQL Database . . . 332
17.1 MySQL Type Mapping . . . 332
17.1.1 String Type Mapping . . . 334

17.1.2 Binary Type Mapping 334
17.1.3 Mixed Automatic/0 Object Id Assignment 335
17.2 MySQL Database Class 336
17.3 MySQL Connection and Connection Factory 339
17.4 MySQL Exceptions 342
17.5 MySQL Limitations 343
17.5.1 Foreign Key Constraints 343
17.6 MySQL Index Definitions 344
17.7 MySQL Stored Procedures 344
18 SQLite Database 347
18.1 SQLite Type Mapping 347
18.1.1 String Type Mapping 349
18.1.2 Binary Type Mapping 350
18.1.3 Incremental BLOB/TEXT I/O 351
18.1.4 Mixed Automatic/Manual Object Id Assignment 357
18.2 SQLite Database Class 357
18.3 SQLite Connection and Connection Factory 360
18.4 Attached SQLite Databases 365
18.5 SQLite Exceptions 367
18.6 SQLite Limitations 368
18.6.1 Query Result Caching 368
18.6.2 Automatic Assignment of Object Ids 368
18.6.3 Foreign Key Constraints 369
18.6.4 Constraint Violations 370
18.6.5 Sharing of Queries 370
18.6.6 Forced Rollback 370
18.6.7 Database Schema Evolution 371
18.7 SQLite Index Definitions 372
19 PostgreSQL Database 373
19.1 PostgreSQL Type Mapping 373
19.1.1 String Type Mapping 375
19.1.2 Binary Type and UUID Mapping 376
19.2 PostgreSQL Database Class 377
19.3 PostgreSQL Connection and Connection Factory 380
19.4 PostgreSQL Exceptions 383
19.5 PostgreSQL Limitations 384
19.5.1 Query Result Caching 384
19.5.2 Foreign Key Constraints 384
19.5.3 Unique Constraint Violations 385
19.5.4 Date-Time Format 385
19.5.5 Timezones 385
19.5.6 NUMERIC Type Support 385
19.5.7 Bulk Operations Support 385

19.6 PostgreSQL Index Definitions . . . 385
19.7 PostgreSQL Stored Procedures and Functions . . . 386
20 Oracle Database . . . 391
20.1 Oracle Type Mapping . . . 391
20.1.1 String Type Mapping . . . 392
20.1.2 Binary Type Mapping . . . 393
20.2 Oracle Database Class . . . 394
20.3 Oracle Connection and Connection Factory . . . 397
20.4 Oracle Exceptions . . . 401
20.5 Oracle Limitations . . . 402
20.5.1 Identifier Truncation . . . 402
20.5.2 Query Result Caching . . . 403
20.5.3 Foreign Key Constraints . . . 403
20.5.4 Unique Constraint Violations . . . 404
20.5.5 Large `FLOAT` and `NUMBER` Types . . . 404
20.5.6 Timezones . . . 404
20.5.7 `LONG` Types . . . 405
20.5.8 LOB Types and By-Value Accessors/Modifiers . . . 405
20.5.9 Database Schema Evolution . . . 405
20.6 Oracle Index Definitions . . . 405
21 Microsoft SQL Server Database . . . 407
21.1 SQL Server Type Mapping . . . 407
21.1.1 String Type Mapping . . . 409
21.1.2 Binary Type and `UNIQUEIDENTIFIER` Mapping . . . 410
21.1.3 `ROWVERSION` Mapping . . . 411
21.1.4 Long String and Binary Types . . . 412
21.2 SQL Server Database Class . . . 413
21.3 SQL Server Connection and Connection Factory . . . 419
21.4 SQL Server Exceptions . . . 423
21.5 SQL Server Limitations . . . 424
21.5.1 Query Result Caching . . . 424
21.5.2 Foreign Key Constraints . . . 425
21.5.3 Unique Constraint Violations . . . 425
21.5.4 Multi-threaded Windows Applications . . . 425
21.5.5 Affected Row Count and DDL Statements . . . 425
21.5.6 Long Data and Auto Object Ids, `ROWVERSION` . . . 425
21.5.7 Long Data and By-Value Accessors/Modifiers . . . 426
21.5.8 Bulk Update and `ROWVERSION` . . . 426
21.6 SQL Server Index Definitions . . . 426
21.7 SQL Server Stored Procedures . . . 427
PART III PROFILES . . . 430
22 Profiles Introduction . . . 431
23 Boost Profile . . . 432

23.1 Smart Pointers Library 432
23.2 Unordered Containers Library 433
23.3 Multi-Index Container Library 434
23.4 Optional Library 435
23.5 Date Time Library 436
23.5.1 MySQL Database Type Mapping 437
23.5.2 SQLite Database Type Mapping 438
23.5.3 PostgreSQL Database Type Mapping 439
23.5.4 Oracle Database Type Mapping 440
23.5.5 SQL Server Database Type Mapping 441
23.6 Uuid Library 442
23.6.1 MySQL Database Type Mapping 442
23.6.2 SQLite Database Type Mapping 442
23.6.3 PostgreSQL Database Type Mapping 443
23.6.4 Oracle Database Type Mapping 443
23.6.5 SQL Server Database Type Mapping 443
24 Qt Profile 444
24.1 Basic Types Library 444
24.1.1 MySQL Database Type Mapping 445
24.1.2 SQLite Database Type Mapping 446
24.1.3 PostgreSQL Database Type Mapping 446
24.1.4 Oracle Database Type Mapping 447
24.1.5 SQL Server Database Type Mapping 447
24.2 Smart Pointers Library 448
24.3 Containers Library 449
24.3.1 Change-Tracking `QList` 450
24.4 Date Time Library 453
24.4.1 MySQL Database Type Mapping 454
24.4.2 SQLite Database Type Mapping 455
24.4.3 PostgreSQL Database Type Mapping 456
24.4.4 Oracle Database Type Mapping 456
24.4.5 SQL Server Database Type Mapping 457

Preface

As more critical aspects of our lives become dependant on software systems, more and more applications are required to save the data they work on in persistent and reliable storage. Database management systems and, in particular, relational database management systems (RDBMS) are commonly used for such storage. However, while the application development techniques and programming languages have evolved significantly over the past decades, the relational database technology in this area stayed relatively unchanged. In particular, this led to the now infamous mismatch between the object-oriented model used by many modern applications and the relational model still used by RDBMS.

While relational databases may be inconvenient to use from modern programming languages, they are still the main choice for many applications due to their maturity, reliability, as well as the availability of tools and alternative implementations.

To allow application developers to utilize relational databases from their object-oriented applications, a technique called object-relational mapping (ORM) is often used. It involves a conversion layer that maps between objects in the application's memory and their relational representation in the database. While the object-relational mapping code can be written manually, automated ORM systems are available for most object-oriented programming languages in use today.

ODB is an ORM system for the C++ programming language. It was designed and implemented with the following main goals:

- Provide a fully-automatic ORM system. In particular, the application developer should not have to manually write any mapping code, neither for persistent classes nor for their data member.
- Provide clean and easy to use object-oriented persistence model and database APIs that support the development of realistic applications for a wide variety of domains.
- Provide a portable and thread-safe implementation. ODB should be written in standard C++ and capable of persisting any standard C++ classes.
- Provide profiles that integrate ODB with type systems of widely-used frameworks and libraries such as Qt and Boost.
- Provide a high-performance and low overhead implementation. ODB should make efficient use of database and application resources.

About This Document

The goal of this manual is to provide you with an understanding of the object persistence model and APIs which are implemented by ODB. As such, this document is intended for C++ application developers and software architects who are looking for a C++ object persistence solution. Prior experience with C++ is required to understand this document. A basic understanding of

relational database systems is advantageous but not expected or required.

More Information

Beyond this manual, you may also find the following sources of information useful:

- ODB Compiler Command Line Manual.
- The `INSTALL` files in the ODB source packages provide build instructions for various platforms.
- The `odb-examples` package contains a collection of examples and a README file with an overview of each example.
- The odb-users mailing list is the place to ask technical questions about ODB. Furthermore, the searchable archives may already have answers to some of your questions.

PART I OBJECT-RELATIONAL MAPPING

Part I describes the essential database concepts, APIs, and tools that together comprise the object-relational mapping for C++ as implemented by ODB. It consists of the following chapters.

1 Introduction

2 Hello World Example

3 Working with Persistent Objects

4 Querying the Database

5 Containers

6 Relationships

7 Value Types

8 Inheritance

10 Views

11 Session

12 Optimistic Concurrency

13 Database Schema Evolution

14 ODB Pragma Language

1 Introduction

ODB is an object-relational mapping (ORM) system for C++. It provides tools, APIs, and library support that allow you to persist C++ objects to a relational database (RDBMS) without having to deal with tables, columns, or SQL and without manually writing any of the mapping code.

ODB is highly flexible and customizable. It can either completely hide the relational nature of the underlying database or expose some of the details as required. For example, you can automatically map basic C++ types to suitable SQL types, generate the relational database schema for your persistent classes, and use simple, safe, and yet powerful object query language instead of SQL. Or you can assign SQL types to individual data members, use the existing database schema, run native SQL `SELECT` queries, and call stored procedures. In fact, at an extreme, ODB can be used as *just* a convenient way to handle results of native SQL queries.

ODB is not a framework. It does not dictate how you should write your application. Rather, it is designed to fit into your style and architecture by only handling object persistence and not interfering with any other functionality. There is no common base type that all persistent classes should derive from nor are there any restrictions on the data member types in persistent classes. Existing classes can be made persistent with a few or no modifications.

ODB has been designed for high performance and low memory overhead. Prepared statements are used to send and receive object state in binary format instead of text which reduces the load on the application and the database server. Extensive caching of connections, prepared statements, and buffers saves time and resources on connection establishment, statement parsing, and memory allocations. For each supported database system the native C API is used instead of ODBC or higher-level wrapper APIs to reduce overhead and provide the most efficient implementation for each database operation. Finally, persistent classes have zero memory overhead. There are no hidden "database" members that each class must have nor are there per-object data structures allocated by ODB.

In this chapter we present a high-level overview of ODB. We will start with the ODB architecture and then outline the workflow of building an application that uses ODB. We will then continue by contrasting the drawbacks of the traditional way of saving C++ objects to relational databases with the benefits of using ODB for object persistence. We conclude the chapter by discussing the C++ standards supported by ODB. The next chapter takes a more hands-on approach and shows the concrete steps necessary to implement object persistence in a simple "Hello World" application.

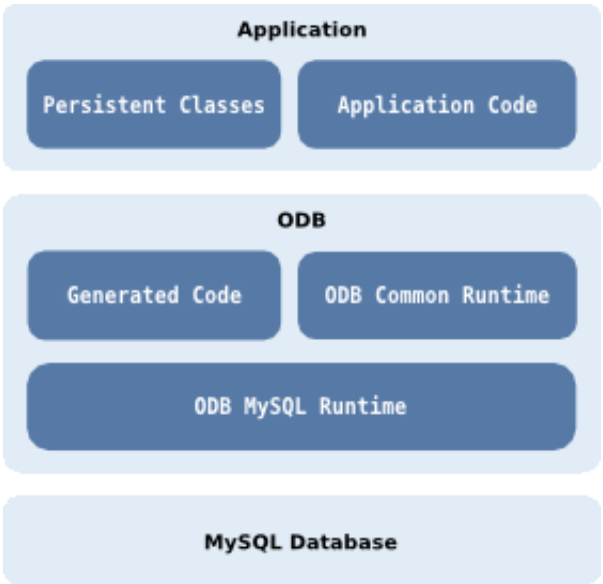
1.1 Architecture and Workflow

From the application developer's perspective, ODB consists of three main components: the ODB compiler, the common runtime library, called `libodb`, and the database-specific runtime libraries, called `libodb-<database>`, where <database> is the name of the database system this runtime is for, for example, `libodb-mysql`. For instance, if the application is going to use the MySQL database for object persistence, then the three ODB components that this application will use are the ODB compiler, `libodb` and `libodb-mysql`.

The ODB compiler generates the database support code for persistent classes in your application. The input to the ODB compiler is one or more C++ header files defining C++ classes that you want to make persistent. For each input header file the ODB compiler generates a set of C++ source files implementing conversion between persistent C++ classes defined in this header and their database representation. The ODB compiler can also generate a database schema file that creates tables necessary to store the persistent classes.

The ODB compiler is a real C++ compiler except that it produces C++ instead of assembly or machine code. In particular, it is not an ad-hoc header pre-processor that is only capable of recognizing a subset of C++. ODB is capable of parsing any standard C++ code.

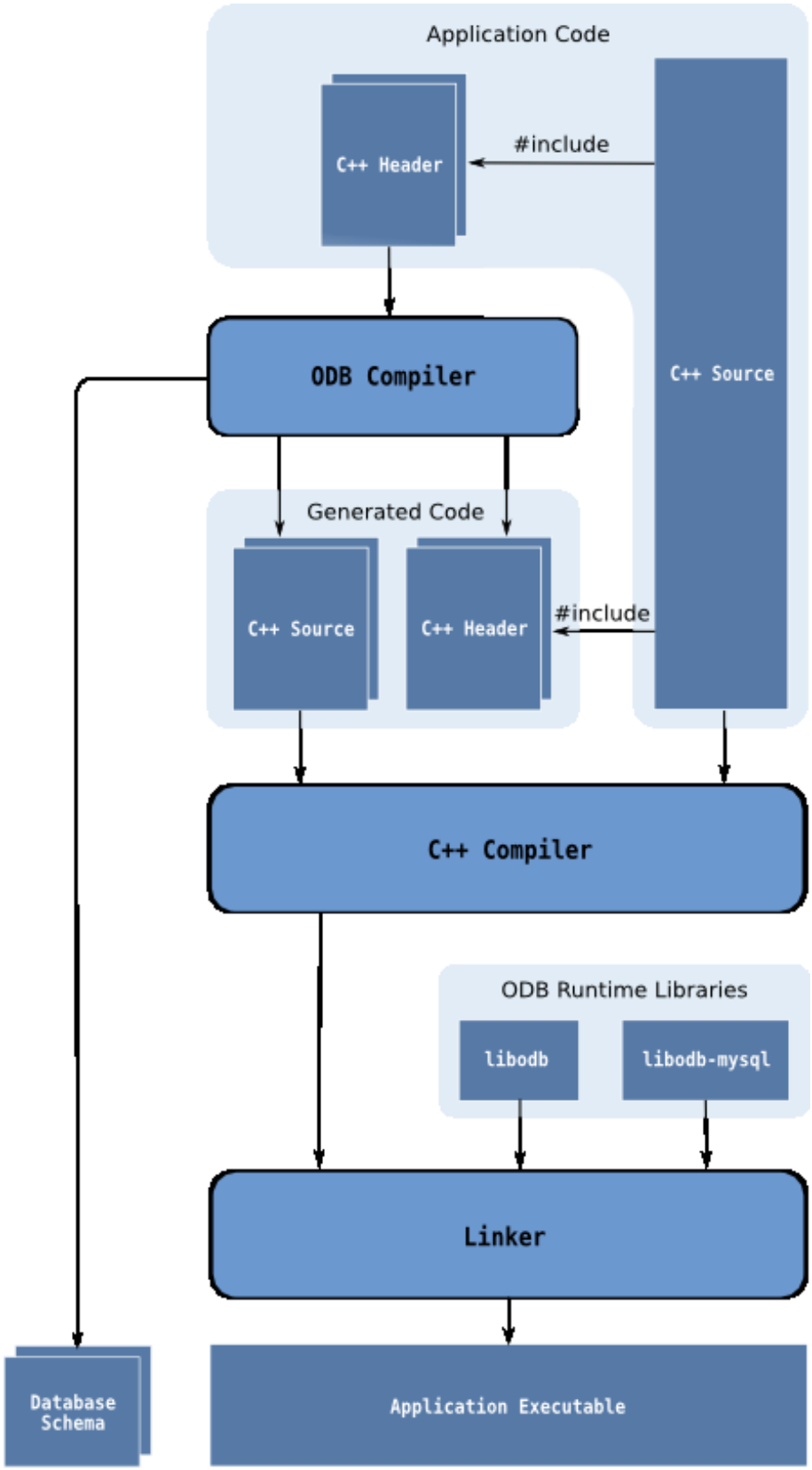
The common runtime library defines database system-independent interfaces that your application can use to manipulate persistent objects. The database-specific runtime library provides implementations of these interfaces for a concrete database as well as other database-specific utilities that are used by the generated code. Normally, the application does not use the database-specific runtime library directly but rather works with it via the common interfaces from `libodb`. The following diagram shows the object persistence architecture of an application that uses MySQL as the underlying database system:



The ODB system also defines two special-purpose languages: the ODB Pragma Language and ODB Query Language. The ODB Pragma Language is used to communicate various properties of persistent classes to the ODB compiler by means of special `#pragma` directives embedded in the C++ header files. It controls aspects of the object-relational mapping such as names of tables and columns that are used for persistent classes and their members or mapping between C++ types and database types.

The ODB Query Language is an object-oriented database query language that can be used to search for objects matching certain criteria. It is modeled after and is integrated into C++ allowing you to write expressive and safe queries that look and feel like ordinary C++.

The use of the ODB compiler to generate database support code adds an additional step to your application build sequence. The following diagram outlines the typical build workflow of an application that uses ODB:


Application Code
C++ Header
#include
C++ Source
ODB Compiler
Generated Code
C++ Source
C++ Header
#include
C++ Compiler
ODB Runtime Libraries
libodb
libodb-mysql
Linker
Database Schema
Application Executable

1.2 Benefits

The traditional way of saving C++ objects to relational databases requires that you manually write code which converts between the database and C++ representations of each persistent class. The actions that such code usually performs include conversion between C++ values and strings or database types, preparation and execution of SQL queries, as well as handling the result sets. Writing this code manually has the following drawbacks:

- **Difficult and time consuming.** Writing database conversion code for any non-trivial application requires extensive knowledge of the specific database system and its APIs. It can also take a considerable amount of time to write and maintain. Supporting multi-threaded applications can complicate this task even further.
- **Suboptimal performance.** Optimal conversion often requires writing large amounts of extra code, such as parameter binding for prepared statements and caching of connections, statements, and buffers. Writing code like this in an ad-hoc manner is often too difficult and time consuming.
- **Database vendor lock-in.** The conversion code is written for a specific database which makes it hard to switch to another database vendor.
- **Lack of type safety.** It is easy to misspell column names or pass incompatible values in SQL queries. Such errors will only be detected at runtime.
- **Complicates the application.** The database conversion code often ends up interspersed throughout the application making it hard to debug, change, and maintain.

In contrast, using ODB for C++ object persistence has the following benefits:

- **Ease of use.** ODB automatically generates database conversion code from your C++ class declarations and allows you to manipulate persistent objects using simple and thread-safe object-oriented database APIs.
- **Concise code.** With ODB hiding the details of the underlying database, the application logic is written using the natural object vocabulary instead of tables, columns and SQL. The resulting code is simpler and thus easier to read and understand.
- **Optimal performance.** ODB has been designed for high performance and low memory overhead. All the available optimization techniques, such as prepared statements and extensive connection, statement, and buffer caching, are used to provide the most efficient implementation for each database operation.
- **Database portability.** Because the database conversion code is automatically generated, it is easy to switch from one database vendor to another. In fact, it is possible to test your application on several database systems before making a choice.
- **Safety.** The ODB object persistence and query APIs are statically typed. You use C++ identifiers instead of strings to refer to object members and the generated code makes sure database and C++ types are compatible. All this helps catch programming errors at compile-time rather than at runtime.

- **Maintainability.** Automatic code generation minimizes the effort needed to adapt the application to changes in persistent classes. The database support code is kept separately from the class declarations and application logic. This makes the application easier to debug and maintain.

Overall, ODB provides an easy to use yet flexible and powerful object-relational mapping (ORM) system for C++. Unlike other ORM implementations for C++ that still require you to write database conversion or member registration code for each persistent class, ODB keeps persistent classes purely declarative. The functional part, the database conversion code, is automatically generated by the ODB compiler from these declarations.

1.3 Supported C++ Standards

ODB provides support for ISO/IEC C++ 1998/2003 (C++98/03), ISO/IEC C++ 2011 (C++11), as well later standards with the majority of the examples in this manual using C++11. The `c++11` example in the `odb-examples` package shows ODB support for various features new in C++11.

2 Hello World Example

In this chapter we will show how to create a simple C++ application that relies on ODB for object persistence using the traditional "Hello World" example. In particular, we will discuss how to declare persistent classes, generate database support code, as well as compile and run our application. We will also learn how to make objects persistent, load, update and delete persistent objects, as well as query the database for persistent objects that match certain criteria. The example also shows how to define and use views, a mechanism that allows us to create projections of persistent objects, database tables, or to handle results of native SQL queries or stored procedure calls.

The code presented in this chapter is based on the `hello` example which can be found in the `odb-examples` package of the ODB distribution.

2.1 Declaring Persistent Classes

In our "Hello World" example we will depart slightly from the norm and say hello to people instead of the world. People in our application will be represented as objects of C++ class `person` which is saved in `person.hxx`:

```
// person.hxx
//

#include <string>

class person
{
public:
  person (const std::string& first,
          const std::string& last,
          unsigned short age);

  const std::string& first () const;
  const std::string& last () const;

  unsigned short age () const;
  void age (unsigned short);

private:
  std::string first_;
  std::string last_;
  unsigned short age_;
};
```

In order not to miss anyone whom we need to greet, we would like to save the `person` objects in a database. To achieve this we declare the `person` class as persistent:

```
// person.hxx
//

#include <string>

#include <odb/core.hxx>     // (1)

#pragma db object           // (2)
class person
{
  ...

private:
  person () {}              // (3)

  friend class odb::access; // (4)

  #pragma db id auto        // (5)
  unsigned long long id_;   // (5)

  std::string first_;
  std::string last_;
  unsigned short age_;
};
```

To be able to save the `person` objects in the database we had to make five changes, marked with (1) to (5), to the original class definition. The first change is the inclusion of the ODB header `<odb/core.hxx>`. This header provides a number of core ODB declarations, such as `odb::access`, that are used to define persistent classes.

The second change is the addition of `db object` pragma just before the class definition. This pragma tells the ODB compiler that the class that follows is persistent. Note that making a class persistent does not mean that all objects of this class will automatically be stored in the database. You would still create ordinary or *transient* instances of this class just as you would before. The difference is that now you can make such transient instances persistent, as we will see shortly.

The third change is the addition of the default constructor. The ODB-generated database support code will use this constructor when instantiating an object from the persistent state. Just as we have done for the `person` class, you can make the default constructor private or protected if you don't want to make it available to the users of your class. Note also that with some limitations it is possible to have a persistent class without the default constructor.

With the fourth change we make the `odb::access` class a friend of our `person` class. This is necessary to make the default constructor and the data members accessible to the database support code. If your class has a public default constructor and either public data members or public accessors and modifiers for the data members, then the `friend` declaration is unnecessary.

The final change adds a data member called `id_` which is preceded by another pragma. In ODB every persistent object normally has a unique, within its class, identifier. Or, in other words, no two persistent instances of the same type have equal identifiers. While it is possible to define a persistent class without an object id, the number of database operations that can be performed on such a class is limited. For our class we use an integer id. The `db id auto` pragma that precedes the `id_` member tells the ODB compiler that the following member is the object's identifier. The `auto` specifier indicates that it is a database-assigned id. A unique id will be automatically generated by the database and assigned to the object when it is made persistent.

In this example we chose to add an identifier because none of the existing members could serve the same purpose. However, if a class already has a member with suitable properties, then it is natural to use that member as an identifier. For example, if our `person` class contained some form of personal identification (SSN in the United States or ID/passport number in other countries), then we could use that as an id. Or, if we stored an email associated with each person, then we could have used that if each person is presumed to have a unique email address.

As another example, consider the following alternative version of the `person` class. Here we use one of the existing data members as id. Also the data members are kept private and are instead accessed via public accessor and modifier functions. Finally, the ODB pragmas are grouped together and are placed after the class definition. They could have also been moved into a separate header leaving the original class completely unchanged (for more information on such a non-intrusive conversion refer to Chapter 14, "ODB Pragma Language").

```
class person
{
public:
  person ();

  const std::string& email () const;
  void email (const std::string&);

  const std::string& get_name () const;
  std::string& set_name ();

  unsigned short getAge () const;
  void setAge (unsigned short);

private:
  std::string email_;
  std::string name_;
  unsigned short age_;
};

#pragma db object(person)
#pragma db member(person::email_) id
```

Now that we have the header file with the persistent class, let's see how we can generate that database support code.

2.2 Generating Database Support Code

The persistent class definition that we created in the previous section was particularly light on any code that could actually do the job and store the person's data to a database. There was no serialization or deserialization code, not even data member registration, that you would normally have to write by hand in other ORM libraries for C++. This is because in ODB code that translates between the database and C++ representations of an object is automatically generated by the ODB compiler.

To compile the `person.hxx` header we created in the previous section and generate the support code for the MySQL database, we invoke the ODB compiler from a terminal (UNIX) or a command prompt (Windows):

```
odb -d mysql --generate-query person.hxx
```

We will use MySQL as the database of choice in the remainder of this chapter, though other supported database systems can be used instead.

If you haven't installed the common ODB runtime library (`libodb`) or installed it into a directory where C++ compilers don't search for headers by default, then you may get the following error:

```
person.hxx:10:24: fatal error: odb/core.hxx: No such file or directory
```

To resolve this you will need to specify the `libodb` headers location with the `-I` preprocessor option, for example:

```
odb -I.../libodb -d mysql --generate-query person.hxx
```

Here `.../libodb` represents the path to the `libodb` directory.

The above invocation of the ODB compiler produces three C++ files: `person-odb.hxx`, `person-odb.ixx`, `person-odb.cxx`. You normally don't use types or functions contained in these files directly. Rather, all you have to do is include `person-odb.hxx` in C++ files where you are performing database operations with classes from `person.hxx` as well as compile `person-odb.cxx` and link the resulting object file to your application.

You may be wondering what the `--generate-query` option is for. It instructs the ODB compiler to generate optional query support code that we will use later in our "Hello World" example. Another option that we will find useful is `--generate-schema`. This option makes the ODB compiler generate a fourth file, `person.sql`, which is the database schema for the persistent classes defined in `person.hxx`:

```
odb -d mysql --generate-query --generate-schema person.hxx
```

The database schema file contains SQL statements that creates tables necessary to store the persistent classes. We will learn how to use it in the next section.

If you would like to see a list of all the available ODB compiler options, refer to the ODB Compiler Command Line Manual.

Now that we have the persistent class and the database support code, the only part that is left is the application code that does something useful with all of this. But before we move on to the fun part, let's first learn how to build and run an application that uses ODB. This way when we have some application code to try, there are no more delays before we can run it.

2.3 Compiling and Running

Assuming that the `main()` function with the application code is saved in `driver.cxx` and the database support code and schema are generated as described in the previous section, to build our application we will first need to compile all the C++ source files and then link them with two ODB runtime libraries.

On UNIX, the compilation part can be done with the following commands (substitute `c++` with your C++ compiler name; for Microsoft Visual Studio setup, see the `odb-examples` package):

```
c++ -c driver.cxx
c++ -c person-odb.cxx
```

Similar to the ODB compilation, if you get an error stating that a header in `odb/` or `odb/mysql` directory is not found, you will need to use the `-I` preprocessor option to specify the location of the common ODB runtime library (`libodb`) and MySQL ODB runtime library (`libodb-mysql`).

Once the compilation is done, we can link the application with the following command:

```
c++ -o driver driver.o person-odb.o -lodb-mysql -lodb
```

Notice that we link our application with two ODB libraries: `libodb` which is a common runtime library and `libodb-mysql` which is a MySQL runtime library (if you use another database, then the name of this library will change accordingly). If you get an error saying that one of these libraries could not be found, then you will need to use the `-L` linker option to specify their locations.

Before we can run our application we need to create a database schema using the generated `person.sql` file. For MySQL we can use the `mysql` client program, for example:

```
mysql --user=odb_test --database=odb_test < person.sql
```

The above command will log in to a local MySQL server as user `odb_test` without a password and use the database named `odb_test`. Beware that after executing this command, all the data stored in the `odb_test` database will be deleted.

Note also that using a standalone generated SQL file is not the only way to create a database schema in ODB. We can also embed the schema directly into our application or use custom schemas that were not generated by the ODB compiler. Refer to Section 3.4, "Database" for details.

Once the database schema is ready, we run our application using the same login and database name:

```
./driver --user odb_test --database odb_test
```

2.4 Making Objects Persistent

Now that we have the infrastructure work out of the way, it is time to see our first code fragment that interacts with the database. In this section we will learn how to make `person` objects persistent:

```
// driver.cxx
//

#include <memory>   // std::unique_ptr
#include <iostream>

#include <odb/database.hxx>
#include <odb/transaction.hxx>

#include <odb/mysql/database.hxx>

#include "person.hxx"
#include "person-odb.hxx"

using namespace std;
using namespace odb::core;

int
main (int argc, char* argv[])
{
  try
  {
    unique_ptr<database> db (new odb::mysql::database (argc, argv));

    unsigned long long john_id, jane_id, joe_id;
```

```
    // Create a few persistent person objects.
    //
    {
      person john ("John", "Doe", 33);
      person jane ("Jane", "Doe", 32);
      person joe ("Joe", "Dirt", 30);

      transaction t (db->begin ());

      // Make objects persistent and save their ids for later use.
      //
      john_id = db->persist (john);
      jane_id = db->persist (jane);
      joe_id = db->persist (joe);

      t.commit ();
    }
  }
  catch (const odb::exception& e)
  {
    cerr << e.what () << endl;
    return 1;
  }
}
```

Let's examine this code piece by piece. At the beginning we include a bunch of headers. After the standard C++ headers we include `<odb/database.hxx>` and `<odb/transaction.hxx>` which define database system-independent `odb::database` and `odb::transaction` interfaces. Then we include `<odb/mysql/database.hxx>` which defines the MySQL implementation of the `database` interface. Finally, we include `person.hxx` and `person-odb.hxx` which define our persistent `person` class.

Then we have two `using namespace` directives. The first one brings in the names from the standard namespace and the second brings in the ODB declarations which we will use later in the file. Notice that in the second directive we use the `odb::core` namespace instead of just `odb`. The former only brings into the current namespace the essential ODB names, such as the `database` and `transaction` classes, without any of the auxiliary objects. This minimizes the likelihood of name conflicts with other libraries. Note also that you should continue using the `odb` namespace when qualifying individual names. For example, you should write `odb::database`, not `odb::core::database`.

Once we are in `main()`, the first thing we do is create the MySQL database object. Notice that this is the last line in `driver.cxx` that mentions MySQL explicitly; the rest of the code works through the common interfaces and is database system-independent. We use the `argc`/`argv` `mysql::database` constructor which automatically extract the database parameters, such as login name, password, database name, etc., from the command line. In your own applications you may prefer to use other `mysql::database` constructors which allow you to pass this informa-

tion directly (Section 17.2, "MySQL Database Class").

Next, we create three `person` objects. Right now they are transient objects, which means that if we terminate the application at this point, they will be gone without any evidence of them ever existing. The next line starts a database transaction. We discuss transactions in detail later in this manual. For now, all we need to know is that all ODB database operations must be performed within a transaction and that a transaction is an atomic unit of work; all database operations performed within a transaction either succeed (committed) together or are automatically undone (rolled back).

Once we are in a transaction, we call the `persist()` database function on each of our `person` objects. At this point the state of each object is saved in the database. However, note that this state is not permanent until and unless the transaction is committed. If, for example, our application crashes at this point, there will still be no evidence of our objects ever existing.

In our case, one more thing happens when we call `persist()`. Remember that we decided to use database-assigned identifiers for our `person` objects. The call to `persist()` is where this assignment happens. Once this function returns, the `id_` member contains this object's unique identifier. As a convenience, the `persist()` function also returns a copy of the object's identifier that it made persistent. We save the returned identifier for each object in a local variable. We will use these identifiers later in the chapter to perform other database operations on our persistent objects.

After we have persisted our objects, it is time to commit the transaction and make the changes permanent. Only after the `commit()` function returns successfully, are we guaranteed that the objects are made persistent. Continuing with the crash example, if our application terminates after the commit for whatever reason, the objects' state in the database will remain intact. In fact, as we will discover shortly, our application can be restarted and load the original objects from the database. Note also that a transaction must be committed explicitly with the `commit()` call. If the `transaction` object leaves scope without the transaction being explicitly committed or rolled back, it will automatically be rolled back. This behavior allows you not to worry about exceptions being thrown within a transaction; if they cross the transaction boundary, the transaction will automatically be rolled back and all the changes made to the database undone.

The final bit of code in our example is the `catch` block that handles the database exceptions. We do this by catching the base ODB exception (Section 3.14, "ODB Exceptions") and printing the diagnostics.

Let's now compile (Section 2.3, "Compiling and Running") and then run our first ODB application:

```
mysql --user=odb_test --database=odb_test < person.sql
./driver --user odb_test --database odb_test
```

Our first application doesn't print anything except for error messages so we can't really tell whether it actually stored the objects' state in the database. While we will make our application more entertaining shortly, for now we can use the `mysql` client to examine the database content. It will also give us a feel for how the objects are stored:

```
mysql --user=odb_test --database=odb_test

Welcome to the MySQL monitor.

mysql> select * from person;

+----+-------+------+-----+
| id | first | last | age |
+----+-------+------+-----+
|  1 | John  | Doe  |  33 |
|  2 | Jane  | Doe  |  32 |
|  3 | Joe   | Dirt |  30 |
+----+-------+------+-----+
3 rows in set (0.00 sec)

mysql> quit
```

Another way to get more insight into what's going on under the hood, is to trace the SQL statements executed by ODB as a result of each database operation. Here is how we can enable tracing just for the duration of our transaction:

```
// Create a few persistent person objects.
//
{
  ...

  transaction t (db->begin ());

  t.tracer (stderr_tracer);

  // Make objects persistent and save their ids for later use.
  //
  john_id = db->persist (john);
  jane_id = db->persist (jane);
  joe_id = db->persist (joe);

  t.commit ();
}
```

With this modification our application now produces the following output:

```
INSERT INTO `person` (`id`,`first`,`last`,`age`) VALUES (?,?,?,?)
INSERT INTO `person` (`id`,`first`,`last`,`age`) VALUES (?,?,?,?)
INSERT INTO `person` (`id`,`first`,`last`,`age`) VALUES (?,?,?,?)
```

Note that we see question marks instead of the actual values because ODB uses prepared statements and sends the data to the database in binary form. For more information on tracing, refer to Section 3.13, "Tracing SQL Statement Execution". In the next section we will see how to access persistent objects from our application.

2.5 Querying the Database for Objects

So far our application doesn't resemble a typical "Hello World" example. It doesn't print anything except for error messages. Let's change that and teach our application to say hello to people from our database. To make it a bit more interesting, let's say hello only to people over 30:

```
// driver.cxx
//

...

int
main (int argc, char* argv[])
{
  try
  {
    ...

    // Create a few persistent person objects.
    //
    {
      ...
    }

    using query = odb::query<person>;
    using result = odb::result<person>;

    // Say hello to those over 30.
    //
    {
      transaction t (db->begin ());

      result r (db->query<person> (query::age > 30));

      for (result::iterator i (r.begin ()); i != r.end (); ++i)
      {
        cout << "Hello, " << i->first () << "!" << endl;
```

```
      }

      t.commit ();
    }
  }
  catch (const odb::exception& e)
  {
    cerr << e.what () << endl;
    return 1;
  }
}
```

The first half of our application is the same as before and is replaced with "..." in the above listing for brevity. Again, let's examine the rest of it piece by piece.

The two `using` declarations create convenient aliases for two template instantiations that will be used a lot in our application. The first is the query type for the `person` objects and the second is the result type for that query.

Then we begin a new transaction and call the `query()` database function. We pass a query expression (`query::age > 30`) which limits the returned objects only to those with the age greater than 30. We also save the result of the query in a local variable.

The next few lines perform a standard for-loop iteration over the result sequence printing hello for every returned person. Then we commit the transaction and that's it. Let's see what this application will print:

```
mysql --user=odb_test --database=odb_test < person.sql
./driver --user odb_test --database odb_test

Hello, John!
Hello, Jane!
```

That looks about right, but how do we know that the query actually used the database instead of just using some in-memory artifacts of the earlier `persist()` calls? One way to test this would be to comment out the first transaction in our application and re-run it without re-creating the database schema. This way the objects that were persisted during the previous run will be returned. Alternatively, we can just re-run the same application without re-creating the schema and notice that we now show duplicate objects:

```
./driver --user odb_test --database odb_test

Hello, John!
Hello, Jane!
Hello, John!
Hello, Jane!
```

What happens here is that the previous run of our application persisted a set of `person` objects and when we re-run the application, we persist another set with the same names but with different ids. When we later run the query, matches from both sets are returned. We can change the line where we print the "Hello" string as follows to illustrate this point:

```
cout << "Hello, " << i->first () << " (" << i->id () << ")!" << endl;
```

If we now re-run this modified program, again without re-creating the database schema, we will get the following output:

```
./driver --user odb_test --database odb_test

Hello, John (1)!
Hello, Jane (2)!
Hello, John (4)!
Hello, Jane (5)!
Hello, John (7)!
Hello, Jane (8)!
```

The identifiers 3, 6, and 9 that are missing from the above list belong to the "Joe Dirt" objects which are not selected by this query.

2.6 Updating Persistent Objects

While making objects persistent and then selecting some of them using queries are two useful operations, most applications will also need to change the object's state and then make these changes persistent. Let's illustrate this by updating Joe's age who just had a birthday:

```
// driver.cxx
//

...

int
main (int argc, char* argv[])
{
  try
  {
    ...

    unsigned long long john_id, jane_id, joe_id;

    // Create a few persistent person objects.
    //
    {
      ...

      // Save object ids for later use.
```

```
      //
      john_id = john.id ();
      jane_id = jane.id ();
      joe_id = joe.id ();
    }

    // Joe Dirt just had a birthday, so update his age.
    //
    {
      transaction t (db->begin ());

      unique_ptr<person> joe (db->load<person> (joe_id));
      joe->age (joe->age () + 1);
      db->update (*joe);

      t.commit ();
    }

    // Say hello to those over 30.
    //
    {
      ...
    }
  }
  catch (const odb::exception& e)
  {
    cerr << e.what () << endl;
    return 1;
  }
}
```

The beginning and the end of the new transaction are the same as the previous two. Once within a transaction, we call the `load()` database function to instantiate a `person` object with Joe's persistent state. We pass Joe's object identifier that we stored earlier when we made this object persistent. While here we use `std::unique_ptr` to manage the returned object, we could have also used another smart pointer, for example `shared_ptr` from C++11 or Boost. For more information on the object lifetime management and the smart pointers that we can use for that, see Section 3.3, "Object and View Pointers".

With the instantiated object in hand we increment the age and call the `update()` function to update the object's state in the database. Once the transaction is committed, the changes are made permanent.

If we now run this application, we will see Joe in the output since he is now over 30:

```
mysql --user=odb_test --database=odb_test < person.sql
./driver --user odb_test --database odb_test

Hello, John!
Hello, Jane!
Hello, Joe!
```

What if we didn't have an identifier for Joe? Maybe this object was made persistent in another run of our application or by another application altogether. Provided that we only have one Joe Dirt in the database, we can use the query facility to come up with an alternative implementation of the above transaction:

```
// Joe Dirt just had a birthday, so update his age. An
// alternative implementation without using the object id.
//
{
  transaction t (db->begin ());

  // Here we know that there can be only one Joe Dirt in our
  // database so we use the query_one() shortcut instead of
  // manually iterating over the result returned by query().
  //
  unique_ptr<person> joe (
    db->query_one<person> (query::first == "Joe" &&
                           query::last == "Dirt"));

  if (joe.get () != 0)
  {
    joe->age (joe->age () + 1);
    db->update (*joe);
  }

  t.commit ();
}
```

2.7 Defining and Using Views

Suppose that we need to gather some basic statistics about the people stored in our database. Things like the total head count, as well as the minimum and maximum ages. One way to do it would be to query the database for all the `person` objects and then calculate this information as we iterate over the query result. While this approach may work fine for our database with just three people in it, it would be very inefficient if we had a large number of objects.

While it may not be conceptually pure from the object-oriented programming point of view, a relational database can perform some computations much faster and much more economically than if we performed the same operations ourselves in the application's process.

To support such cases ODB provides the notion of views. An ODB view is a C++ `class` that embodies a light-weight, read-only projection of one or more persistent objects or database tables or the result of a native SQL query execution or stored procedure call.

Some of the common applications of views include loading a subset of data members from objects or columns database tables, executing and handling results of arbitrary SQL queries, including aggregate queries, as well as joining multiple objects and/or database tables using object relationships or custom join conditions.

While you can find a much more detailed description of views in Chapter 10, "Views", here is how we can define the `person_stat` view that returns the basic statistics about the `person` objects:

```
#pragma db view object(person)
struct person_stat
{
  #pragma db column("count(" + person::id_ + ")")
  std::size_t count;

  #pragma db column("min(" + person::age_ + ")")
  unsigned short min_age;

  #pragma db column("max(" + person::age_ + ")")
  unsigned short max_age;
};
```

Normally, to get the result of a view we use the same `query()` function as when querying the database for an object. Here, however, we are executing an aggregate query which always returns exactly one element. Therefore, instead of getting the result instance and then iterating over it, we can use the shortcut `query_value()` function. Here is how we can load and print our statistics using the view we have just created:

```
// Print some statistics about all the people in our database.
//
{
  transaction t (db->begin ());

  // The result of this query always has exactly one element.
  //
  person_stat ps (db->query_value<person_stat> ());

  cout << "count  : " << ps.count << endl
       << "min age: " << ps.min_age << endl
       << "max age: " << ps.max_age << endl;

  t.commit ();
}
```

If we now add the `person_stat` view to the `person.hxx` header, the above transaction to `driver.cxx`, as well as re-compile and re-run our example, then we will see the following additional lines in the output:

```
count  : 3
min age: 31
max age: 33
```

2.8 Deleting Persistent Objects

The last operation that we will discuss in this chapter is deleting the persistent object from the database. The following code fragment shows how we can delete an object given its identifier:

```
// John Doe is no longer in our database.
//
{
  transaction t (db->begin ());
  db->erase<person> (john_id);
  t.commit ();
}
```

To delete John from the database we start a transaction, call the `erase()` database function with John's object id, and commit the transaction. After the transaction is committed, the erased object is no longer persistent.

If we don't have an object id handy, we can use queries to find and delete the object:

```
// John Doe is no longer in our database. An alternative
// implementation without using the object id.
//
{
  transaction t (db->begin ());

  // Here we know that there can be only one John Doe in our
  // database so we use the query_one() shortcut again.
  //
  unique_ptr<person> john (
    db->query_one<person> (query::first == "John" &&
                           query::last == "Doe"));

  if (john.get () != 0)
    db->erase (*john);

  t.commit ();
}
```

2.9 Changing Persistent Classes

When the definition of a transient C++ class is changed, for example by adding or deleting a data member, we don't have to worry about any existing instances of this class not matching the new definition. After all, to make the class changes effective we have to restart the application and none of the transient instances will survive this.

Things are not as simple for persistent classes. Because they are stored in the database and therefore survive application restarts, we have a new problem: what happens to the state of existing objects (which correspond to the old definition) once we change our persistent class?

The problem of working with old objects, called *database schema evolution*, is a complex issue and ODB provides comprehensive support for handling it. While this support is covered in detail in Chapter 13, "Database Schema Evolution", let us consider a simple example that should give us a sense of the functionality provided by ODB in this area.

Suppose that after using our `person` persistent class for some time and creating a number of databases containing its instances, we realized that for some people we also need to store their middle name. If we go ahead and just add the new data member, everything will work fine with new databases. Existing databases, however, have a table that does not correspond to the new class definition. Specifically, the generated database support code now expects there to be a column to store the middle name. But such a column was never created in the old databases.

ODB can automatically generate SQL statements that will migrate old databases to match the new class definitions. But first, we need to enable schema evolution support by defining a version for our object model:

```
// person.hxx
//

#pragma db model version(1, 1)

class person
{
  ...

  std::string first_;
  std::string last_;
  unsigned short age_;
};
```

The first number in the `version` pragma is the base model version. This is the lowest version we will be able to migrate from. The second number is the current model version. Since we haven't made any changes yet to our persistent class, both of these values are `1`.

Next we need to re-compile our `person.hxx` header file with the ODB compiler, just as we did before:

```
odb -d mysql --generate-query --generate-schema person.hxx
```

If we now look at the list of files produced by the ODB compiler, we will notice a new file: `person.xml`. This is a changelog file where the ODB compiler keeps track of the database changes corresponding to our class changes. Note that this file is automatically maintained by the ODB compiler and all we have to do is keep it around between re-compilations.

Now we are ready to add the middle name to our `person` class. We also give it a default value (empty string) which is what will be assigned to existing objects in old databases. Notice that we have also incremented the current version:

```
// person.hxx
//

#pragma db model version(1, 2)

class person
{
  ...

  std::string first_;

  #pragma db default("")
  std::string middle_;

  std::string last_;
  unsigned short age_;
};
```

If we now recompile the `person.hxx` header again, we will see two extra generated files: `person-002-pre.sql` and `person-002-post.sql`. These two files contain schema migration statements from version `1` to version `2`. Similar to schema creation, schema migration statements can also be embedded into the generated C++ code.

`person-002-pre.sql` and `person-002-post.sql` are the pre and post schema migration files. To migrate one of our old databases, we first execute the pre migration file:

```
mysql --user=odb_test --database=odb_test < person-002-pre.sql
```

Between the pre and post schema migrations we can run data migration code, if required. At this stage, we can both access the old and store the new data. In our case we don't need any data migration code since we assigned the default value to the middle name for all the existing objects.

To finish the migration process we execute the post migration statements:

```
mysql --user=odb_test --database=odb_test < person-002-post.sql
```

2.10 Working with Multiple Databases

Accessing multiple databases (that is, data stores) is simply a matter of creating multiple `odb::<db>::database` instances representing each database. For example:

```
odb::mysql::database db1 ("john", "secret", "test_db1");
odb::mysql::database db2 ("john", "secret", "test_db2");
```

Some database systems also allow attaching multiple databases to the same instance. A more interesting question is how we access multiple database systems (that is, database implementations) from the same application. For example, our application may need to store some objects in a remote MySQL database and others in a local SQLite file. Or, our application may need to be able to store its objects in a database system that is selected by the user at runtime.

ODB provides comprehensive multi-database support that ranges from tight integration with specific database systems to being able to write database-agnostic code and loading individual database systems support dynamically. While all these aspects are covered in detail in Chapter 16, "Multi-Database Support", in this section we will get a taste of this functionality by extending our "Hello World" example to be able to store its data either in MySQL or PostgreSQL (other database systems supported by ODB can be added in a similar manner).

The first step in adding multi-database support is to re-compile our `person.hxx` header to generate database support code for additional database systems:

```
odb --multi-database dynamic -d common -d mysql -d pgsql \
--generate-query --generate-schema person.hxx
```

The `--multi-database` ODB compiler option turns on multi-database support. For now it is not important what the `dynamic` value that we passed to this option means, but if you are curious, see Chapter 16. The result of this command are three sets of generated files: `person-odb.?xx` (common interface; corresponds to the `common` database), `person-odb-mysql.?xx` (MySQL support code), and `person-odb-pgsql.?xx` (PostgreSQL support code). There are also two schema files: `person-mysql.sql` and `person-pgsql.sql`.

The only part that we need to change in `driver.cxx` is how we create the database instance. Specifically, this line:

```
unique_ptr<database> db (new odb::mysql::database (argc, argv));
```

Now our example is capable of storing its data either in MySQL or PostgreSQL so we need to somehow allow the caller to specify which database we must use. To keep things simple, we will make the first command line argument specify the database system we must use while the rest will contain the database-specific options which we will pass to the `odb::<db>::database` constructor as before. Let's put all this logic into a separate function which we will call `create_database()`. Here is what the beginning of our modified `driver.cxx` will look like (the remainder is unchanged):

```
// driver.cxx
//

#include <string>
#include <memory>   // std::unique_ptr
#include <iostream>

#include <odb/database.hxx>
#include <odb/transaction.hxx>

#include <odb/mysql/database.hxx>
#include <odb/pgsql/database.hxx>

#include "person.hxx"
#include "person-odb.hxx"

using namespace std;
using namespace odb::core;

unique_ptr<database>
create_database (int argc, char* argv[])
{
  unique_ptr<database> r;

  if (argc < 2)
  {
    cerr << "error: database system name expected" << endl;
    return r;
  }

  string db (argv[1]);

  if (db == "mysql")
    r.reset (new odb::mysql::database (argc, argv));
  else if (db == "pgsql")
    r.reset (new odb::pgsql::database (argc, argv));
  else
    cerr << "error: unknown database system " << db << endl;

  return r;
```

```
}

int
main (int argc, char* argv[])
{
  try
  {
    unique_ptr<database> db (create_database (argc, argv));

    if (db.get () == 0)
      return 1; // Diagnostics has already been issued.

    ...
```

And that's it. The only thing left is to build and run our example:

```
c++ -c driver.cxx
c++ -c person-odb.cxx
c++ -c person-odb-mysql.cxx
c++ -c person-odb-pgsql.cxx
c++ -o driver driver.o person-odb.o person-odb-mysql.o \
person-odb-pgsql.o -lodb-mysql -lodb-pgsql -lodb
```

Here is how we can access a MySQL database:

```
mysql --user=odb_test --database=odb_test < person-mysql.sql
./driver mysql --user odb_test --database odb_test
```

Or a PostgreSQL database:

```
psql --user=odb_test --dbname=odb_test -f person-pgsql.sql
./driver pgsql --user odb_test --database odb_test
```

2.11 Summary

This chapter presented a very simple application which, nevertheless, exercised all of the core database functions: `persist()`, `query()`, `load()`, `update()`, and `erase()`. We also saw that writing an application that uses ODB involves the following steps:

1. Declare persistent classes in header files.
2. Compile these headers to generate database support code.
3. Link the application with the generated code and two ODB runtime libraries.

Do not be concerned if, at this point, much appears unclear. The intent of this chapter is to give you only a general idea of how to persist C++ objects with ODB. We will cover all the details throughout the remainder of this manual.

3 Working with Persistent Objects

The previous chapters gave us a high-level overview of ODB and showed how to use it to store C++ objects in a database. In this chapter we will examine the ODB object persistence model as well as the core database APIs in greater detail. We will start with basic concepts and terminology in Section 3.1 and Section 3.3 and continue with the discussion of the `odb::database` class in Section 3.4, transactions in Section 3.5, and connections in Section 3.6. The remainder of this chapter deals with the core database operations and concludes with the discussion of ODB exceptions.

In this chapter we will continue to use and expand the `person` persistent class that we have developed in the previous chapter.

3.1 Concepts and Terminology

The term *database* can refer to three distinct things: a general notion of a place where an application stores its data, a software implementation for managing this data (for example MySQL), and, finally, some database software implementations may manage several data stores which are usually distinguished by name. This name is also commonly referred to as a database.

In this manual, when we use the word *database*, we refer to the first meaning above, for example, "The `update()` function saves the object's state to the database." The term Database Management System (DBMS) is often used to refer to the second meaning of the word database. In this manual we will use the term *database system* for short, for example, "Database system-independent application code." Finally, to distinguish the third meaning from the other two, we will use the term *database name*, for example, "The second option specifies the database name that the application should use to store its data."

In C++ there is only one notion of a type and an instance of a type. For example, a fundamental type, such as `int`, is, for the most part, treated the same as a user defined class type. However, when it comes to persistence, we have to place certain restrictions and requirements on certain C++ types that can be stored in the database. As a result, we divide persistent C++ types into two groups: *object types* and *value types*. An instance of an object type is called an *object* and an instance of a value type — a *value*.

An object is an independent entity. It can be stored, updated, and deleted in the database independent of other objects. Normally, an object has an identifier, called *object id*, that is unique among all instances of an object type within a database. In contrast, a value can only be stored in the database as part of an object and doesn't have its own unique identifier.

An object consists of data members which are either values (Chapter 7, "Value Types"), pointers to other objects (Chapter 6, "Relationships"), or containers of values or pointers to other objects (Chapter 5, "Containers"). Pointers to other objects and containers can be viewed as special kinds

of values since they also can only be stored in the database as part of an object. Static data members are not stored in the database.

An object type is a C++ class. Because of this one-to-one relationship, we will use terms *object type* and *object class* interchangeably. In contrast, a value type can be a fundamental C++ type, such as `int` or a class type, such as `std::string`. If a value consists of other values, then it is called a *composite value* and its type — a *composite value type* (Section 7.2, "Composite Value Types"). Otherwise, the value is called *simple value* and its type — a *simple value type* (Section 7.1, "Simple Value Types"). Note that the distinction between simple and composite values is conceptual rather than representational. For example, `std::string` is a simple value type because conceptually string is a single value even though the representation of the string class may contain several data members each of which could be considered a value. In fact, the same value type can be viewed (and mapped) as both simple and composite by different applications.

While not strictly necessary in a purely object-oriented application, practical considerations often require us to only load a subset of an object's data members or a combination of members from several objects. We may also need to factor out some computations to the relational database instead of performing them in the application's process. To support such requirements ODB distinguishes a third kind of C++ types, called *views* (Chapter 10, "Views"). An ODB view is a C++ `class` that embodies a light-weight, read-only projection of one or more persistent objects or database tables or the result of a native SQL query execution.

Understanding how all these concepts map to the relational model will hopefully make these distinctions clearer. In a relational database an object type is mapped to a table and a value type is mapped to one or more columns. A simple value type is mapped to a single column while a composite value type is mapped to several columns. An object is stored as a row in this table and a value is stored as one or more cells in this row. A simple value is stored in a single cell while a composite value occupies several cells. A view is not a persistent entity and it is not stored in the database. Rather, it is a data structure that is used to capture a single row of an SQL query result.

Going back to the distinction between simple and composite values, consider a date type which has three integer members: year, month, and day. In one application it can be considered a composite value and each member will get its own column in a relational database. In another application it can be considered a simple value and stored in a single column as a number of days from some predefined date.

Until now, we have been using the term *persistent class* to refer to object classes. We will continue to do so even though a value type can also be a class. The reason for this asymmetry is the subordinate nature of value types when it comes to database operations. Remember that values are never stored directly but rather as part of an object that contains them. As a result, when we say that we want to make a C++ class persistent or persist an instance of a class in the database, we invariably refer to an object class rather than a value class.

Normally, you would use object types to model real-world entities, things that have their own identity. For example, in the previous chapter we created a `person` class to model a person, which is a real-world entity. Name and age, which we used as data members in our `person` class are clearly values. It is hard to think of age 31 or name "Joe" as having their own identities.

A good test to determine whether something is an object or a value, is to consider if other objects might reference it. A person is clearly an object because it can be referred to by other objects such as a spouse, an employer, or a bank. On the other hand, a person's age or name is not something that other objects would normally refer to.

Also, when an object represents a real entity, it is easy to choose a suitable object id. For example, for a person there is an established notion of an identifier (SSN, student id, passport number, etc). Another alternative is to use a person's email address as an identifier.

Note, however, that these are only guidelines. There could be good reasons to make something that would normally be a value an object. Consider, for example, a database that stores a vast number of people. Many of the `person` objects in this database have the same names and surnames and the overhead of storing them in every object may negatively affect the performance. In this case, we could make the first name and last name each an object and only store pointers to these objects in the `person` class.

An instance of a persistent class can be in one of two states: *transient* and *persistent*. A transient instance only has a representation in the application's memory and will cease to exist when the application terminates, unless it is explicitly made persistent. In other words, a transient instance of a persistent class behaves just like an instance of any ordinary C++ class. A persistent instance has a representation in both the application's memory and the database. A persistent instance will remain even after the application terminates unless and until it is explicitly deleted from the database.

3.2 Declaring Persistent Objects and Values

To make a C++ class a persistent object class we declare it as such using the `db object` pragma, for example:

```
#pragma db object
class person
{
  ...
};
```

The other pragma that we often use is `db id` which designates one of the data members as an object id, for example:

```
#pragma db object
class person
{
  ...

  #pragma db id
  unsigned long long id_;
};
```

The object id can be of a simple or composite (Section 7.2.1, "Composite Object Ids") value type. This type should be default-constructible, copy-constructible, and copy-assignable. It is also possible to declare a persistent class without an object id, however, such a class will have limited functionality (Section 14.1.6, "`no_id`").

The above two pragmas are the minimum required to declare a persistent class with an object id. Other pragmas can be used to fine-tune the database-related properties of a class and its members (Chapter 14, "ODB Pragma Language").

Normally, a persistent class should define the default constructor. The generated database support code uses this constructor when instantiating an object from the persistent state. If we add the default constructor only for the database support code, then we can make it private provided we also make the `odb::access` class, defined in the `<odb/core.hxx>` header, a friend of this object class. For example:

```
#include <odb/core.hxx>

#pragma db object
class person
{
  ...

private:
  friend class odb::access;
  person () {}
};
```

It is also possible to have an object class without the default constructor. However, in this case, the database operations will only be able to load the persistent state into an existing instance (Section 3.9, "Loading Persistent Objects", Section 4.4, "Query Result").

The ODB compiler also needs access to the non-transient (Section 14.4.11, "`transient`") data members of a persistent class. The ODB compiler can access such data members directly if they are public. It can also do so if they are private or protected and the `odb::access` class is declared a friend of the object type. For example:

```
#include <odb/core.hxx>

#pragma db object
class person
{
  ...

private:
  friend class odb::access;
  person () {}

  #pragma db id
  unsigned long long id_;

  std::string name_;
};
```

If data members are not accessible directly, then the ODB compiler will try to automatically find suitable accessor and modifier functions. To accomplish this, the ODB compiler will try to lookup common accessor and modifier names derived from the data member name. Specifically, for the `name_` data member in the above example, the ODB compiler will look for accessor functions with names: `get_name()`, `getName()`, `getname()`, and just `name()` as well as for modifier functions with names: `set_name()`, `setName()`, `setname()`, and just `name()`. You can also add support for custom name derivations with the `--accessor-regex` and `--modifier-regex` ODB compiler options. Refer to the ODB Compiler Command Line Manual for details on these options. The following example illustrates automatic accessor and modifier discovery:

```
#pragma db object
class person
{
public:
  person () {}

  ...

  unsigned long long id () const;
  void id (unsigned long long);

  const std::string& get_name () const;
  std::string& set_name ();

private:
  #pragma db id
  unsigned long long id_; // Uses id() for access.

  std::string name_; // Uses get_name()/set_name() for access.
};
```

Finally, if a data member is not directly accessible and the ODB compiler was unable to discover suitable accessor and modifier functions, then we can provide custom accessor and modifier expressions using the `db get` and `db set` pragmas. For more information on custom accessor and modifier expressions refer to Section 14.4.5, "`get/set/access`".

Data members of a persistent class can also be split into separately-loaded and/or separately-updated sections. For more information on this functionality, refer to Chapter 9, "Sections".

You may be wondering whether we also have to declare value types as persistent. We don't need to do anything special for simple value types such as `int` or `std::string` since the ODB compiler knows how to map them to suitable database types and how to convert between the two. On the other hand, if a simple value is unknown to the ODB compiler then we will need to provide the mapping to the database type and, possibly, the code to convert between the two. For more information on how to achieve this refer to the `db type` pragma description in Section 14.3.1, "`type`".

Similar to object classes, composite value types have to be explicitly declared as persistent using the `db value` pragma, for example:

```
#pragma db value
class name
{
  ...

  std::string first_;
  std::string last_;
};
```

Note that a composite value cannot have a data member designated as an object id since, as we have discussed earlier, values do not have a notion of identity. A composite value type also doesn't have to define the default constructor, unless it is used as an element of a container. The ODB compiler uses the same mechanisms to access data members in composite value types as in object types. Composite value types are discussed in more detail in Section 7.2, "Composite Value Types".

3.3 Object and View Pointers

As we have seen in the previous chapter, some database operations create dynamically allocated instances of persistent classes and return pointers to these instances. As we will see in later chapters, pointers are also used to establish relationships between objects (Chapter 6, "Relationships") as well as to cache persistent objects in a session (Chapter 11, "Session"). While in most cases you won't need to deal with pointers to views, it is possible to a obtain a dynamically allocated instance of a view using the `result_iterator::load()` function (Section 4.4, "Query Results").

By default, all these mechanisms use raw pointers to return objects and views as well as to pass and cache objects. This is normally sufficient for applications that have simple object lifetime requirements and do not use sessions or object relationships. In particular, a dynamically allocated object or view that is returned as a raw pointer from a database operation can be assigned to a smart pointer of our choice, for example `std::auto_ptr` (C++98/03 only), `std::unique_ptr` from C++11, or `shared_ptr` from C++11 or Boost.

However, to avoid any possibility of a mistake, such as forgetting to use a smart pointer for a returned object or view, as well as to simplify the use of more advanced ODB functionality, such as sessions and bidirectional object relationships, it is recommended that you use smart pointers with the sharing semantics as object pointers. The `shared_ptr` smart pointer from C++11 or Boost is a good default choice. However, if sharing is not required and sessions are not used, then `std::unique_ptr` or `std::auto_ptr` can be used just as well.

ODB provides several mechanisms for changing the object or view pointer type. To specify the pointer type on the per object or per view basis we can use the `db pointer` pragma, for example:

```
#pragma db object pointer(std::shared_ptr)
class person
{
  ...
};
```

We can also specify the default pointer for a group of objects or views at the namespace level:

```
#pragma db namespace pointer(std::shared_ptr)
namespace accounting
{
  #pragma db object
  class employee
  {
    ...
  };

  #pragma db object
  class employer
  {
    ...
  };
}
```

Finally, we can use the `--default-pointer` option to specify the default pointer for the whole file. Refer to the ODB Compiler Command Line Manual for details on this option's argument. The typical usage is shown below:

```
--default-pointer std::shared_ptr
```

An alternative to this method with the same effect is to specify the default pointer for the global namespace:

```
#pragma db namespace() pointer(std::shared_ptr)
```

Note that we can always override the default pointer specified at the namespace level or with the command line option using the `db pointer` object or view pragma. For example:

```
#pragma db object pointer(std::shared_ptr)
namespace accounting
{
  #pragma db object
  class employee
  {
    ...
  };

  #pragma db object pointer(std::unique_ptr)
  class employer
  {
    ...
  };
}
```

Refer to Section 14.1.2, "`pointer` (object)", Section 14.2.4, "`pointer` (view)", and Section 14.5.1, "`pointer` (namespace)" for more information on these mechanisms.

Built-in support that is provided by the ODB runtime library allows us to use `std::shared_ptr` (C++11), `std::unique_ptr` (C++11), or `std::auto_ptr` (C++98/03 only) as pointer types. Plus, ODB profile libraries, that are available for commonly used frameworks and libraries (such as Boost and Qt), provide support for smart pointers found in these frameworks and libraries (Part III, "Profiles"). It is also easy to add support for our own smart pointers, as described in Section 6.5, "Using Custom Smart Pointers".

3.4 Database

Before an application can make use of persistence services offered by ODB, it has to create a database class instance. A database instance is the representation of the place where the application stores its persistent objects. We create a database instance by instantiating one of the database system-specific classes. For example, `odb::mysql::database` would be such a class for the MySQL database system. We will also normally pass a database name as an argument to the class' constructor. The following code fragment shows how we can create a database instance for the MySQL database system:

```
#include <odb/database.hxx>
#include <odb/mysql/database.hxx>

unique_ptr<odb::database> db (
  new odb::mysql::database (
    "test_user"     // database login name
    "test_password" // database password
    "test_database" // database name
    ));
```

The `odb::database` class is a common interface for all the database system-specific classes provided by ODB. You would normally work with the database instance via this interface unless there is a specific functionality that your application depends on and which is only exposed by a particular system's `database` class. You will need to include the `<odb/database.hxx>` header file to make this class available in your application.

The `odb::database` interface defines functions for starting transactions and manipulating persistent objects. These are discussed in detail in the remainder of this chapter as well as the next chapter which is dedicated to the topic of querying the database for persistent objects. For details on the system-specific `database` classes, refer to Part II, "Database Systems".

Before we can persist our objects, the corresponding database schema has to be created in the database. The schema contains table definitions and other relational database artifacts that are used to store the state of persistent objects in the database.

There are several ways to create the database schema. The easiest is to instruct the ODB compiler to generate the corresponding schema from the persistent classes (`--generate-schema` option). The ODB compiler can generate the schema as a standalone SQL file, embedded into the generated C++ code, or as a separate C++ source file (`--schema-format` option). If we are using the SQL file to create the database schema, then this file should be executed, normally only once, before the application is started.

Alternatively, if the schema is embedded directly into the generated code or produced as a separate C++ source file, then we can use the `odb::schema_catalog` class to create it in the database from within our application, for example:

```
#include <odb/schema-catalog.hxx>

odb::transaction t (db->begin ());
odb::schema_catalog::create_schema (*db);
t.commit ();
```

Refer to the next section for information on the `odb::transaction` class. The complete version of the above code fragment is available in the `schema/embedded` example in the `odb-examples` package.

The `odb::schema_catalog` class has the following interface. You will need to include the `<odb/schema-catalog.hxx>` header file to make this class available in your application.

```
namespace odb
{
  class schema_catalog
  {
  public:
    static void
    create_schema (database&,
                   const std::string& name = "",
                   bool drop = true);

    static void
    drop_schema (database&, const std::string& name = "");

    static bool
    exists (database_id, const std::string& name = "");

    static bool
    exists (const database&, const std::string& name = "")
  };
}
```

The first argument to the `create_schema()` function is the database instance that we would like to create the schema in. The second argument is the schema name. By default, the ODB compiler generates all embedded schemas with the default schema name (empty string). However, if your application needs to have several separate schemas, you can use the `--schema-name` ODB compiler option to assign custom schema names and then use these names as a second argument to `create_schema()`. By default, `create_schema()` will also delete all the database objects (tables, indexes, etc.) if they exist prior to creating the new ones. You can change this behavior by passing `false` as the third argument. The `drop_schema()` function allows you to delete all the database objects without creating the new ones.

If the schema is not found, the `create_schema()` and `drop_schema()` functions throw the `odb::unknown_schema` exception. You can use the `exists()` function to check whether a schema for the specified database and with the specified name exists in the catalog. Note also that the `create_schema()` and `drop_schema()` functions should be called within a transaction.

ODB also provides support for database schema evolution. Similar to schema creation, schema migration statements can be generated either as standalone SQL files or embedded into the generated C++ code. For more information on schema evolution support, refer to Chapter 13, "Database Schema Evolution".

Finally, we can also use a custom database schema with ODB. This approach can work similarly to the standalone SQL file described above except that the database schema is hand-written or produced by another program. Or we could execute custom SQL statements that create the schema directly from our application. To map persistent classes to custom database schemas, ODB provides a wide range of mapping customization pragmas, such as `db table`, `db column`, and `db type` (Chapter 14, "ODB Pragma Language"). For sample code that shows how to perform such mapping for various C++ constructs, refer to the `schema/custom` example in the `odb-examples` package.

3.5 Transactions

A transaction is an atomic, consistent, isolated and durable (ACID) unit of work. Database operations can only be performed within a transaction and each thread of execution in an application can have only one active transaction at a time.

By atomicity we mean that when it comes to making changes to the database state within a transaction, either all the changes are applied or none at all. Consider, for example, a transaction that transfers funds between two objects representing bank accounts. If the debit function on the first object succeeds but the credit function on the second fails, the transaction is rolled back and the database state of the first object remains unchanged.

By consistency we mean that a transaction must take all the objects stored in the database from one consistent state to another. For example, if a bank account object must reference a person object as its owner and we forget to set this reference before making the object persistent, the transaction will be rolled back and the database will remain unchanged.

By isolation we mean that the changes made to the database state during a transaction are only visible inside this transaction until and unless it is committed. Using the above example with the bank transfer, the results of the debit operation performed on the first object is not visible to other transactions until the credit operation is successfully completed and the transaction is committed.

By durability we mean that once the transaction is committed, the changes that it made to the database state are permanent and will survive failures such as an application crash. From now on the only way to alter this state is to execute and commit another transaction.

A transaction is started by calling either the `database::begin()` or `connection::begin()` function. The returned transaction handle is stored in an instance of the `odb::transaction` class. You will need to include the `<odb/transaction.hxx>` header file to make this class available in your application. For example:

```
#include <odb/transaction.hxx>

transaction t (db.begin ())

// Perform database operations.

t.commit ();
```

The `odb::transaction` class has the following interface:

```
namespace odb
{
  class transaction
  {
  public:
    using database_type = odb::database;
    using connection_type = odb::connection;

    explicit
    transaction (transaction_impl*, bool make_current = true);

    transaction ();

    void
    reset (transaction_impl*, bool make_current = true);

    void
    commit ();

    void
    rollback ();

    database_type&
    database ();

    connection_type&
    connection ();

    bool
    finilized () const;

  public:
    static bool
    has_current ();

    static transaction&
    current ();

    static void
    current (transaction&);
```

```
    static bool
    reset_current ();

    // Callback API.
    //
  public:
    ...
  };
}
```

The `commit()` function commits a transaction and `rollback()` rolls it back. Unless the transaction has been *finalized*, that is, explicitly committed or rolled back, the destructor of the `transaction` class will automatically roll it back when the transaction instance goes out of scope. If we try to commit or roll back a finalized transaction, the `odb::transaction_already_finalized` exception is thrown.

The `database()` accessor returns the database this transaction is working on. Similarly, the `connection()` accessor returns the database connection this transaction is on (Section 3.6, "Connections").

The static `current()` accessor returns the currently active transaction for this thread. If there is no active transaction, this function throws the `odb::not_in_transaction` exception. We can check whether there is a transaction in effect in this thread using the `has_current()` static function.

The `make_current` argument in the `transaction` constructor as well as the static `current()` modifier and `reset_current()` function give us additional control over the nomination of the currently active transaction. If we pass `false` as the `make_current` argument, then the newly created transaction will not automatically be made the active transaction for this thread. Later, we can use the static `current()` modifier to set this transaction as the active transaction. The `reset_current()` static function clears the currently active transaction. Together, these mechanisms allow for more advanced use cases, such as multiplexing two or more transactions on the same thread. For example:

```
transaction t1 (db1.begin ());        // Active transaction.
transaction t2 (db2.begin (), false); // Not active.

// Perform database operations on db1.

transaction::current (t2);            // Deactivate t1, activate t2.

// Perform database operations on db2.

transaction::current (t1);            // Switch back to t1.

// Perform some more database operations on db1.
```

```
t1.commit ();

transaction::current (t2);            // Switch to t2.

// Perform some more database operations on db2.

t2.commit ();
```

The `reset()` modifier allows us to reuse the same `transaction` instance to complete several database transactions. Similar to the destructor, `reset()` will roll the current transaction back if it hasn't been finalized. The default `transaction` constructor creates a finalized transaction which can later be initialized using `reset()`. The `finilized()` accessor can be used to check whether the transaction has been finalized. Here is how we can use this functionality to commit the current transaction and start a new one every time a certain number of database operations has been performed:

```
transaction t (db.begin ());

for (size_t i (0); i < n; ++i)
{
  // Perform a database operation, such as persist an object.

  // Commit the current transaction and start a new one after
  // every 100 operations.
  //
  if (i % 100 == 0)
  {
    t.commit ();
    t.reset (db.begin ());
  }
}

t.commit ();
```

For more information on the transaction callback support, refer to Section 15.1, "Transaction Callbacks".

Note that in the above discussion of atomicity, consistency, isolation, and durability, all of those guarantees only apply to the object's state in the database as opposed to the object's state in the application's memory. It is possible to roll a transaction back but still have changes from this transaction in the application's memory. An easy way to avoid this potential inconsistency is to instantiate persistent objects only within the transaction scope. Consider, for example, these two implementations of the same transaction:

```
void
update_age (database& db, person& p)
{
  transaction t (db.begin ());

  p.age (p.age () + 1);
  db.update (p);

  t.commit ();
}
```

In the above implementation, if the `update()` call fails and the transaction is rolled back, the state of the `person` object in the database and the state of the same object in the application's memory will differ. Now consider an alternative implementation which only instantiates the `person` object for the duration of the transaction:

```
void
update_age (database& db, unsigned long long id)
{
  transaction t (db.begin ());

  unique_ptr<person> p (db.load<person> (id));
  p.age (p.age () + 1);
  db.update (p);

  t.commit ();
}
```

Of course, it may not always be possible to write the application in this style. Oftentimes we need to access and modify the application's state of persistent objects out of transactions. In this case it may make sense to try to roll back the changes made to the application state if the transaction was rolled back and the database state remains unchanged. One way to do this is to re-load the object's state from the database, for example:

```
void
update_age (database& db, person& p)
{
  try
  {
    transaction t (db.begin ());

    p.age (p.age () + 1);
    db.update (p);

    t.commit ();
  }
  catch (...)
  {
    transaction t (db.begin ());
```

```
    db.load (p.id (), p);
    t.commit ();

    throw;
  }
}
```

See also Section 15.1, "Transaction Callbacks" for an alternative approach.

3.6 Connections

The `odb::connection` class represents a connection to the database. Normally, you wouldn't work with connections directly but rather let the ODB runtime obtain and release connections as needed. However, certain use cases may require obtaining a connection manually. For completeness, this section describes the `connection` class and discusses some of its use cases. You may want to skip this section if you are reading through the manual for the first time.

Similar to `odb::database`, the `odb::connection` class is a common interface for all the database system-specific classes provided by ODB. For details on the system-specific `connection` classes, refer to Part II, "Database Systems".

To make the `odb::connection` class available in your application you will need to include the `<odb/connection.hxx>` header file. The `odb::connection` class has the following interface:

```
namespace odb
{
  class connection
  {
  public:
    using database_type = odb::database;

    transaction
    begin () = 0;

    unsigned long long
    execute (const char* statement);

    unsigned long long
    execute (const std::string& statement);

    unsigned long long
    execute (const char* statement, std::size_t length);

    database_type&
    database ();
```

```
  };

  using connection_ptr = details::shared_ptr<connection>;
}
```

The `begin()` function is used to start a transaction on the connection. The `execute()` functions allow us to execute native database statements on the connection. Their semantics are equivalent to the `database::execute()` functions (Section 3.12, "Executing Native SQL Statements") except that they can be legally called outside a transaction. Finally, the `database()` accessor returns a reference to the `odb::database` instance to which this connection corresponds.

To obtain a connection we call the `database::connection()` function. The connection is returned as `odb::connection_ptr`, which is an implementation-specific smart pointer with the shared pointer semantics. This, in particular, means that the connection pointer can be copied and returned from functions. Once the last instance of `connection_ptr` pointing to the same connection is destroyed, the connection is returned to the `database` instance. The following code fragment shows how we can obtain, use, and release a connection:

```
using namespace odb::core;

database& db = ...
connection_ptr c (db.connection ());

// Temporarily disable foreign key constraints.
//
c->execute ("SET FOREIGN_KEY_CHECKS = 0");

// Start a transaction on this connection.
//
transaction t (c->begin ());
...
t.commit ();

// Restore foreign key constraints.
//
c->execute ("SET FOREIGN_KEY_CHECKS = 1");

// When 'c' goes out of scope, the connection is returned to 'db'.
```

Some of the use cases which may require direct manipulation of connections include out-of-transaction statement execution, such as the execution of connection configuration statements, the implementation of a connection-per-thread policy, and making sure that a set of transactions is executed on the same connection.

3.7 Error Handling and Recovery

ODB uses C++ exceptions to report database operation errors. Most ODB exceptions signify *hard* errors or errors that cannot be corrected without some intervention from the application. For example, if we try to load an object with an unknown object id, the `odb::object_not_persistent` exception is thrown. Our application may be able to correct this error, for instance, by obtaining a valid object id and trying again. The hard errors and corresponding ODB exceptions that can be thrown by each database function are described in the remainder of this chapter with Section 3.14, "ODB Exceptions" providing a quick reference for all the ODB exceptions.

The second group of ODB exceptions signify *soft* or *recoverable* errors. Such errors are temporary failures which normally can be corrected by simply re-executing the transaction. ODB defines three such exceptions: `odb::connection_lost`, `odb::timeout`, and `odb::deadlock`. All recoverable ODB exceptions are derived from the common `odb::recoverable` base exception which can be used to handle all the recoverable conditions with a single `catch` block.

The `odb::connection_lost` exception is thrown if a connection to the database is lost in the middle of a transaction. In this situation the transaction is aborted but it can be re-tried without any changes. Similarly, the `odb::timeout` exception is thrown if one of the database operations or the whole transaction has timed out. Again, in this case the transaction is aborted but can be re-tried as is.

If two or more transactions access or modify more than one object and are executed concurrently by different applications or by different threads within the same application, then it is possible that these transactions will try to access objects in an incompatible order and deadlock. The canonical example of a deadlock are two transactions in which the first has modified `object1` and is waiting for the second transaction to commit its changes to `object2` so that it can also update `object2`. At the same time the second transaction has modified `object2` and is waiting for the first transaction to commit its changes to `object1` because it also needs to modify `object1`. As a result, none of the two transactions can be completed.

The database system detects such situations and automatically aborts the waiting operation in one of the deadlocked transactions. In ODB this translates to the `odb::deadlock` recoverable exception being thrown from one of the database functions.

The following code fragment shows how to handle the recoverable exceptions by restarting the affected transaction:

```
const unsigned short max_retries = 5;

for (unsigned short retry_count (0); ; retry_count++)
{
```

```
  try
  {
    transaction t (db.begin ());

    ...

    t.commit ();
    break;
  }
  catch (const odb::recoverable& e)
  {
    if (retry_count > max_retries)
      throw retry_limit_exceeded (e.what ());
    else
      continue;
  }
}
```

3.8 Making Objects Persistent

A newly created instance of a persistent class is transient. We use the `database::persist()` function template to make a transient instance persistent. This function has four overloaded versions with the following signatures:

```
template <typename T>
typename object_traits<T>::id_type
persist (const T& object);

template <typename T>
typename object_traits<T>::id_type
persist (const object_traits<T>::const_pointer_type& object);

template <typename T>
typename object_traits<T>::id_type
persist (T& object);

template <typename T>
typename object_traits<T>::id_type
persist (const object_traits<T>::pointer_type& object);
```

Here and in the rest of the manual, `object_traits<T>::pointer_type` and `object_traits<T>::const_pointer_type` denote the unrestricted and constant object pointer types (Section 3.3, "Object and View Pointers"), respectively. Similarly, `object_traits<T>::id_type` denotes the object id type. The `odb::object_traits` template is part of the database support code generated by the ODB compiler.

The first `persist()` function expects a constant reference to an instance being persisted. The second function expects a constant object pointer. Both of these functions can only be used on objects with application-assigned object ids (Section 14.4.2, "`auto`").

The second and third `persist()` functions are similar to the first two except that they operate on unrestricted references and object pointers. If the identifier of the object being persisted is assigned by the database, these functions update the id member of the passed instance with the assigned value. All four functions return the object id of the newly persisted object.

If the database already contains an object of this type with this identifier, the `persist()` functions throw the `odb::object_already_persistent` exception. This should never happen for database-assigned object ids as long as the number of objects persisted does not exceed the value space of the id type.

When calling the `persist()` functions, we don't need to explicitly specify the template type since it will be automatically deduced from the argument being passed. The following example shows how we can call these functions:

```
person john ("John", "Doe", 33);
shared_ptr<person> jane (new person ("Jane", "Doe", 32));

transaction t (db.begin ());

db.persist (john);
unsigned long long jane_id (db.persist (jane));

t.commit ();

cerr << "Jane's id: " << jane_id << endl;
```

Notice that in the above code fragment we have created instances that we were planning to make persistent before starting the transaction. Likewise, we printed Jane's id after we have committed the transaction. As a general rule, you should avoid performing operations within the transaction scope that can be performed before the transaction starts or after it terminates. An active transaction consumes both your application's resources, such as a database connection, as well as the database server's resources, such as object locks. By following the above rule you make sure these resources are released and made available to other threads in your application and to other applications as soon as possible.

Some database systems support persisting multiple objects with a single underlying statement execution which can result in significantly improved performance. For such database systems ODB provides bulk `persist()` functions. For details, refer to Section 15.3, "Bulk Database Operations".

3.9 Loading Persistent Objects

Once an object is made persistent, and you know its object id, it can be loaded by the application using the `database::load()` function template. This function has two overloaded versions with the following signatures:

```
template <typename T>
typename object_traits<T>::pointer_type
load (const typename object_traits<T>::id_type& id);

template <typename T>
void
load (const typename object_traits<T>::id_type& id, T& object);
```

Given an object id, the first function allocates a new instance of the object class in the dynamic memory, loads its state from the database, and returns the pointer to the new instance. The second function loads the object's state into an existing instance. Both functions throw `odb::object_not_persistent` if there is no object of this type with this id in the database.

When we call the first `load()` function, we need to explicitly specify the object type. We don't need to do this for the second function because the object type will be automatically deduced from the second argument, for example:

```
transaction t (db.begin ());

unique_ptr<person> jane (db.load<person> (jane_id));

db.load (jane_id, *jane);

t.commit ();
```

In certain situations it may be necessary to reload the state of an object from the database. While this is easy to achieve using the second `load()` function, ODB provides the `database::reload()` function template that has a number of special properties. This function has two overloaded versions with the following signatures:

```
template <typename T>
void
reload (T& object);

template <typename T>
void
reload (const object_traits<T>::pointer_type& object);
```

The first `reload()` function expects an object reference, while the second expects an object pointer. Both functions expect the id member in the passed object to contain a valid object identifier and, similar to `load()`, both will throw `odb::object_not_persistent` if there is no object of this type with this id in the database.

The first special property of `reload()` compared to the `load()` function is that it does not interact with the session's object cache (Section 11.1, "Object Cache"). That is, if the object being reloaded is already in the cache, then it will remain there after `reload()` returns. Similarly, if the object is not in the cache, then `reload()` won't put it there either.

The second special property of the `reload()` function only manifests itself when operating on an object with the optimistic concurrency model. In this case, if the states of the object in the application memory and in the database are the same, then no reloading will occur. For more information on optimistic concurrency, refer to Chapter 12, "Optimistic Concurrency".

If we don't know for sure whether an object with a given id is persistent, we can use the `find()` function instead of `load()`, for example:

```
template <typename T>
typename object_traits<T>::pointer_type
find (const typename object_traits<T>::id_type& id);

template <typename T>
bool
find (const typename object_traits<T>::id_type& id, T& object);
```

If an object with this id is not found in the database, the first `find()` function returns a `NULL` pointer while the second function leaves the passed instance unmodified and returns `false`.

If we don't know the object id, then we can use queries to find the object (or objects) matching some criteria (Chapter 4, "Querying the Database"). Note, however, that loading an object's state using its identifier can be significantly faster than executing a query.

3.10 Updating Persistent Objects

If a persistent object has been modified, we can store the updated state in the database using the `database::update()` function template. This function has three overloaded versions with the following signatures:

```
template <typename T>
void
update (const T& object);

template <typename T>
void
update (const object_traits<T>::const_pointer_type& object);
```

```
  template <typename T>
  void
  update (const object_traits<T>::pointer_type& object);
```

The first `update()` function expects an object reference, while the other two expect object pointers. If the object passed to one of these functions does not exist in the database, `update()` throws the `odb::object_not_persistent` exception (but see a note on optimistic concurrency below).

Below is an example of the funds transfer that we talked about in the earlier section on transactions. It uses the hypothetical `bank_account` persistent class:

```
void
transfer (database& db,
          unsigned long long from_acc,
          unsigned long long to_acc,
          unsigned int amount)
{
  bank_account from, to;

  transaction t (db.begin ());

  db.load (from_acc, from);

  if (from.balance () < amount)
    throw insufficient_funds ();

  db.load (to_acc, to);

  to.balance (to.balance () + amount);
  from.balance (from.balance () - amount);

  db.update (to);
  db.update (from);

  t.commit ();
}
```

The same can be accomplished using dynamically allocated objects and the `update()` function with object pointer argument, for example:

```
transaction t (db.begin ());

shared_ptr<bank_account> from (db.load<bank_account> (from_acc));

if (from->balance () < amount)
  throw insufficient_funds ();

shared_ptr<bank_account> to (db.load<bank_account> (to_acc));
```

```
to->balance (to->balance () + amount);
from->balance (from->balance () - amount);

db.update (to);
db.update (from);

t.commit ();
```

If any of the `update()` functions are operating on a persistent class with the optimistic concurrency model, then they will throw the `odb::object_changed` exception if the state of the object in the database has changed since it was last loaded into the application memory. Furthermore, for such classes, `update()` no longer throws the `object_not_persistent` exception if there is no such object in the database. Instead, this condition is treated as a change of object state and `object_changed` is thrown instead. For a more detailed discussion of optimistic concurrency, refer to Chapter 12, "Optimistic Concurrency".

In ODB, persistent classes, composite value types, as well as individual data members can be declared read-only (see Section 14.1.4, "`readonly` (object)", Section 14.3.6, "`readonly` (composite value)", and Section 14.4.12, "`readonly` (data member)").

If an individual data member is declared read-only, then any changes to this member will be ignored when updating the database state of an object using any of the above `update()` functions. A `const` data member is automatically treated as read-only. If a composite value is declared read-only then all its data members are treated as read-only.

If the whole object is declared read-only then the database state of this object cannot be changed. Calling any of the above `update()` functions for such an object will result in a compile-time error.

Similar to `persist()`, for database systems that support this functionality, ODB provides bulk `update()` functions. For details, refer to Section 15.3, "Bulk Database Operations".

3.11 Deleting Persistent Objects

To delete a persistent object's state from the database we use the `database::erase()` or `database::erase_query()` function templates. If the application still has an instance of the erased object, this instance becomes transient. The `erase()` function has the following overloaded versions:

```
template <typename T>
void
erase (const T& object);

template <typename T>
void
```

```
erase (const object_traits<T>::const_pointer_type& object);

template <typename T>
void
erase (const object_traits<T>::pointer_type& object);

template <typename T>
void
erase (const typename object_traits<T>::id_type& id);
```

The first `erase()` function uses an object itself, in the form of an object reference, to delete its state from the database. The next two functions accomplish the same result but using object pointers. Note that all three functions leave the passed object unchanged. It simply becomes transient. The last function uses the object id to identify the object to be deleted. If the object does not exist in the database, then all four functions throw the `odb::object_not_persistent` exception (but see a note on optimistic concurrency below).

We have to specify the object type when calling the last `erase()` function. The same is unnecessary for the first three functions because the object type will be automatically deduced from their arguments. The following example shows how we can call these functions:

```
person& john = ...
shared_ptr<jane> jane = ...
unsigned long long joe_id = ...

transaction t (db.begin ());

db.erase (john);
db.erase (jane);
db.erase<person> (joe_id);

t.commit ();
```

If any of the `erase()` functions except the last one are operating on a persistent class with the optimistic concurrency model, then they will throw the `odb::object_changed` exception if the state of the object in the database has changed since it was last loaded into the application memory. Furthermore, for such classes, `erase()` no longer throws the `object_not_persistent` exception if there is no such object in the database. Instead, this condition is treated as a change of object state and `object_changed` is thrown instead. For a more detailed discussion of optimistic concurrency, refer to Chapter 12, "Optimistic Concurrency".

Similar to `persist()` and `update()`, for database systems that support this functionality, ODB provides bulk `erase()` functions. For details, refer to Section 15.3, "Bulk Database Operations".

The `erase_query()` function allows us to delete the state of multiple objects matching certain criteria. It uses the query expression of the `database::query()` function (Chapter 4, "Querying the Database") and, because the ODB query facility is optional, it is only available if the `--generate-query` ODB compiler option was specified. The `erase_query()` function has the following overloaded versions:

```
template <typename T>
unsigned long long
erase_query ();

template <typename T>
unsigned long long
erase_query (const odb::query<T>&);
```

The first `erase_query()` function is used to delete the state of all the persistent objects of a given type stored in the database. The second function uses the passed query instance to only delete the state of objects matching the query criteria. Both functions return the number of objects erased. When calling the `erase_query()` function, we have to explicitly specify the object type we are erasing. For example:

```
using query = odb::query<person>;

transaction t (db.begin ());

db.erase_query<person> (query::last == "Doe" && query::age < 30);

t.commit ();
```

Unlike the `query()` function, when calling `erase_query()` we cannot use members from pointed-to objects in the query expression. However, we can still use a member corresponding to a pointer as an ordinary object member that has the id type of the pointed-to object (Chapter 6, "Relationships"). This allows us to compare object ids as well as test the pointer for `NULL`. As an example, the following transaction makes sure that all the `employee` objects that reference an `employer` object that is about to be deleted are deleted as well. Here we assume that the `employee` class contains a pointer to the `employer` class. Refer to Chapter 6, "Relationships" for complete definitions of these classes.

```
using query = odb::query<employee>;

transaction t (db.begin ());

employer& e = ... // Employer object to be deleted.

db.erase_query<employee> (query::employer == e.id ());
db.erase (e);

t.commit ();
```

3.12 Executing Native SQL Statements

In some situations we may need to execute native SQL statements instead of using the object-oriented database API described above. For example, we may want to tune the database schema generated by the ODB compiler or take advantage of a feature that is specific to the database system we are using. The `database::execute()` function, which has three overloaded versions, provides this functionality:

```
unsigned long long
execute (const char* statement);

unsigned long long
execute (const std::string& statement);

unsigned long long
execute (const char* statement, std::size_t length)
```

The first `execute()` function expects the SQL statement as a zero-terminated C-string. The last version expects the explicit statement length as the second argument and the statement itself may contain `'\0'` characters, for example, to represent binary data, if the database system supports it. All three functions return the number of rows that were affected by the statement. For example:

```
transaction t (db.begin ());

db.execute ("DROP TABLE test");
db.execute ("CREATE TABLE test (n INT PRIMARY KEY)");

t.commit ();
```

While these functions must always be called within a transaction, it may be necessary to execute a native statement outside a transaction. This can be done using the `connection::execute()` functions as described in Section 3.6, "Connections".

3.13 Tracing SQL Statement Execution

Oftentimes it is useful to understand what SQL statements are executed as a result of high-level database operations. For example, we can use this information to figure out why certain transactions don't produce desired results or why they take longer than expected.

While this information can usually be obtained from the database logs, ODB provides an application-side SQL statement tracing support that is both more convenient and finer-grained. For example, in a typical situation that calls for tracing we would like to see the SQL statements executed as a result of a specific transaction. While it may be difficult to extract such a subset of statements from the database logs, it is easy to achieve with ODB tracing support:

```
transaction t (db.begin ());
t.tracer (stderr_tracer);

...

t.commit ();
```

ODB allows us to specify a tracer on the database, connection, and transaction levels. If specified for the database, then all the statements executed on this database will be traced. On the other hand, if a tracer is specified for the connection, then only the SQL statements executed on this connection will be traced. Similarly, a tracer specified for a transaction will only show statements that are executed as part of this transaction. All three classes (`odb::database`, `odb::connection`, and `odb::transaction`) provide the identical tracing API:

```
  void
  tracer (odb::tracer&);

  void
  tracer (odb::tracer*);

  odb::tracer*
  tracer () const;
```

The first two `tracer()` functions allow us to set the tracer object with the second one allowing us to clear the current tracer by passing a `NULL` pointer. The last `tracer()` function allows us to get the current tracer object. It returns a `NULL` pointer if there is no tracer in effect. Note that the tracing API does not manage the lifetime of the tracer object. The tracer should be valid for as long as it is being used. Furthermore, the tracing API is not thread-safe. Trying to set a tracer from multiple threads simultaneously will result in undefined behavior.

The `odb::tracer` class defines a callback interface that can be used to create custom tracer implementations. The `odb::stderr_tracer` and `odb::stderr_full_tracer` are built-in tracer implementations provided by the ODB runtime. They both print SQL statements being executed to the standard error stream. The full tracer, in addition to tracing statement executions, also traces their preparations and deallocations. One situation where the full tracer can be particularly useful is if a statement (for example a custom query) contains a syntax error. In this case the error will be detected during preparation and, as a result, the statement will never be executed. The only way to see such a statement is by using the full tracing.

The `odb::tracer` class is defined in the `<odb/tracer.hxx>` header file which you will need to include in order to make this class available in your application. The `odb::tracer` interface provided the following callback functions:

```
namespace odb
{
  class tracer
  {
```

```
  public:
    virtual void
    prepare (connection&, const statement&);

    virtual void
    execute (connection&, const statement&);

    virtual void
    execute (connection&, const char* statement) = 0;

    virtual void
    deallocate (connection&, const statement&);
  };
}
```

The `prepare()` and `deallocate()` functions are called when a prepared statement is created and destroyed, respectively. The first `execute()` function is called when a prepared statement is executed while the second one is called when a normal statement is executed. The default implementations for the `prepare()` and `deallocate()` functions do nothing while the first `execute()` function calls the second one passing the statement text as the second argument. As a result, if all you are interested in are the SQL statements being executed, then you only need to override the second `execute()` function.

In addition to the common `odb::tracer` interface, each database runtime provides a database-specific version as `odb::<database>::tracer`. It has exactly the same interface as the common version except that the `connection` and `statement` types are database-specific, which gives us access to additional, database-specific information.

As an example, consider a more elaborate, PostgreSQL-specific tracer implementation. Here we rely on the fact that the PostgreSQL ODB runtime uses names to identify prepared statements and this information can be obtained from the `odb::pgsql::statement` object:

```
#include <odb/pgsql/tracer.hxx>
#include <odb/pgsql/database.hxx>
#include <odb/pgsql/connection.hxx>
#include <odb/pgsql/statement.hxx>

class pgsql_tracer: public odb::pgsql::tracer
{
  virtual void
  prepare (odb::pgsql::connection& c, const odb::pgsql::statement& s)
  {
    cerr << c.database ().db () << ": PREPARE " << s.name ()
         << " AS " << s.text () << endl;
  }

  virtual void
  execute (odb::pgsql::connection& c, const odb::pgsql::statement& s)
```

```
  {
    cerr << c.database ().db () << ": EXECUTE " << s.name () << endl;
  }

  virtual void
  execute (odb::pgsql::connection& c, const char* statement)
  {
    cerr << c.database ().db () << ": " << statement << endl;
  }

  virtual void
  deallocate (odb::pgsql::connection& c, const odb::pgsql::statement& s)
  {
    cerr << c.database ().db () << ": DEALLOCATE " << s.name () << endl;
  }
};
```

Note also that you can only set a database-specific tracer object using a database-specific database instance, for example:

```
pgsql_tracer tracer;

odb::database& db = ...;
db.tracer (tracer); // Compile error.

odb::pgsql::database& db = ...;
db.tracer (tracer); // Ok.
```

3.14 ODB Exceptions

In the previous sections we have already mentioned some of the exceptions that can be thrown by the database functions. In this section we will discuss the ODB exception hierarchy and document all the exceptions that can be thrown by the common ODB runtime.

The root of the ODB exception hierarchy is the abstract `odb::exception` class. This class derives from `std::exception` and has the following interface:

```
namespace odb
{
  struct exception: std::exception
  {
    virtual const char*
    what () const throw () = 0;
  };
}
```

Catching this exception guarantees that we will catch all the exceptions thrown by ODB. The `what()` function returns a human-readable description of the condition that triggered the exception.

The concrete exceptions that can be thrown by ODB are presented in the following listing:

```
namespace odb
{
  struct null_pointer: exception
  {
    virtual const char*
    what () const throw ();
  };

  // Transaction exceptions.
  //
  struct already_in_transaction: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct not_in_transaction: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct transaction_already_finalized: exception
  {
    virtual const char*
    what () const throw ();
  };

  // Session exceptions.
  //
  struct already_in_session: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct not_in_session: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct session_required: exception
  {
```

```
    virtual const char*
    what () const throw ();
  };

  // Database operations exceptions.
  //
  struct recoverable: exception
  {
  };

  struct connection_lost: recoverable
  {
    virtual const char*
    what () const throw ();
  };

  struct timeout: recoverable
  {
    virtual const char*
    what () const throw ();
  };

  struct deadlock: recoverable
  {
    virtual const char*
    what () const throw ();
  };

  struct object_not_persistent: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct object_already_persistent: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct object_changed: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct result_not_cached: exception
  {
    virtual const char*
    what () const throw ();
  };
```

```
struct database_exception: exception
{
};

// Polymorphism support exceptions.
//
struct abstract_class: exception
{
  virtual const char*
  what () const throw ();
};

struct no_type_info: exception
{
  virtual const char*
  what () const throw ();
};

// Prepared query support exceptions.
//
struct prepared_already_cached: exception
{
  const char*
  name () const;

  virtual const char*
  what () const throw ();
};

struct prepared_type_mismatch: exception
{
  const char*
  name () const;

  virtual const char*
  what () const throw ();
};

// Schema catalog exceptions.
//
struct unknown_schema: exception
{
  const std::string&
  name () const;

  virtual const char*
  what () const throw ();
};

struct unknown_schema_version: exception
```

```
  {
    schema_version
    version () const;

    virtual const char*
    what () const throw ();
  };

  // Section exceptions.
  //
  struct section_not_loaded: exception
  {
    virtual const char*
    what () const throw ();
  };

  struct section_not_in_object: exception
  {
    virtual const char*
    what () const throw ();
  };

  // Bulk operation exceptions.
  //
  struct multiple_exceptions: exception
  {
    ...

    virtual const char*
    what () const throw ();
  };
}
```

The `null_pointer` exception is thrown when a pointer to a persistent object declared non-`NULL` with the `db not_null` or `db value_not_null` pragma has the `NULL` value. See Chapter 6, "Relationships" for details.

The next three exceptions (`already_in_transaction`, `not_in_transaction`, `transaction_already_finalized`) are thrown by the `odb::transaction` class and are discussed in Section 3.5, "Transactions".

The next two exceptions (`already_in_session`, and `not_in_session`) are thrown by the `odb::session` class and are discussed in Chapter 11, "Session".

The `session_required` exception is thrown when ODB detects that correctly loading a bidirectional object relationship requires a session but one is not used. See Section 6.2, "Bidirectional Relationships" for more information on this exception.

The `recoverable` exception serves as a common base for all the recoverable exceptions, which are: `connection_lost`, `timeout`, and `deadlock`. The `connection_lost` exception is thrown when a connection to the database is lost. Similarly, the `timeout` exception is thrown if one of the database operations or the whole transaction has timed out. The `deadlock` exception is thrown when a transaction deadlock is detected by the database system. These exceptions can be thrown by any database function. See Section 3.7, "Error Handling and Recovery" for details.

The `object_already_persistent` exception is thrown by the `persist()` database function. See Section 3.8, "Making Objects Persistent" for details.

The `object_not_persistent` exception is thrown by the `load()`, `update()`, and `erase()` database functions. Refer to Section 3.9, "Loading Persistent Objects", Section 3.10, "Updating Persistent Objects", and Section 3.11, "Deleting Persistent Objects" for more information.

The `object_changed` exception is thrown by the `update()` database function and certain `erase()` database functions when operating on objects with the optimistic concurrency model. See Chapter 12, "Optimistic Concurrency" for details.

The `result_not_cached` exception is thrown by the query result class. Refer to Section 4.4, "Query Result" for details.

The `database_exception` exception is a base class for all database system-specific exceptions that are thrown by the database system-specific runtime library. Refer to Part II, "Database Systems" for more information.

The `abstract_class` exception is thrown by the database functions when we attempt to persist, update, load, or erase an instance of a polymorphic abstract class. For more information on abstract classes, refer to Section 14.1.3, "`abstract`".

The `no_type_info` exception is thrown by the database functions when we attempt to persist, update, load, or erase an instance of a polymorphic class for which no type information is present in the application. This normally means that the generated database support code for this class has not been linked (or dynamically loaded) into the application or the discriminator value has not been mapped to a persistent class. For more information on polymorphism support, refer to Section 8.2, "Polymorphism Inheritance".

The `prepared_already_cached` exception is thrown by the `cache_query()` function if a prepared query with the specified name is already cached. The `prepared_type_mismatch` exception is thrown by the `lookup_query()` function if the specified prepared query object type or parameters type does not match the one in the cache. Refer to Section 4.5, "Prepared Queries" for details.

The `unknown_schema` exception is thrown by the `odb::schema_catalog` class if a schema with the specified name is not found. Refer to Section 3.4, "Database" for details. The `unknown_schema_version` exception is thrown by the `schema_catalog` functions that deal with database schema evolution if the passed or current version is unknow. Refer to Chapter 13, "Database Schema Evolution" for details.

The `section_not_loaded` exception is thrown if we attempt to update an object section that hasn't been loaded. The `section_not_in_object` exception is thrown if the section instance being loaded or updated does not belong to the corresponding object. See Chapter 9, "Sections" for more information on these exceptions.

The `multiple_exceptions` exception is thrown by the bulk API functions. Refer to Section 15.3, "Bulk Database Operations" for details.

The `odb::exception` class is defined in the `<odb/exception.hxx>` header file. All the concrete ODB exceptions are defined in `<odb/exceptions.hxx>` which also includes `<odb/exception.hxx>`. Normally you don't need to include either of these two headers because they are automatically included by `<odb/database.hxx>`. However, if the source file that handles ODB exceptions does not include `<odb/database.hxx>`, then you will need to explicitly include one of these headers.

4 Querying the Database

If we don't know the identifiers of the objects that we are looking for, we can use queries to search the database for objects matching certain criteria. The ODB query facility is optional and we need to explicitly request the generation of the necessary database support code with the `--generate-query` ODB compiler option.

ODB provides a flexible query API that offers two distinct levels of abstraction from the database system query language such as SQL. At the high level we are presented with an easy to use yet powerful object-oriented query language, called ODB Query Language. This query language is modeled after and is integrated into C++ allowing us to write expressive and safe queries that look and feel like ordinary C++. We have already seen examples of these queries in the introductory chapters. Below is another, more interesting, example:

```
using query = odb::query<person>;
using result = odb::result<person>;

unsigned short age;
query q (query::first == "John" && query::age < query::_ref (age));

for (age = 10; age < 100; age += 10)
{
  result r (db.query<person> (q));
  ...
}
```

At the low level, queries can be written as predicates using the database system-native query language such as the `WHERE` predicate from the SQL `SELECT` statement. This language will be referred to as native query language. At this level ODB still takes care of converting query parameters from C++ to the database system format. Below is the re-implementation of the above example using SQL as the native query language:

```
query q ("first = 'John' AND age = " + query::_ref (age));
```

Note that at this level we lose the static typing of query expressions. For example, if we wrote something like this:

```
query q (query::first == 123 && query::agee < query::_ref (age));
```

We would get two errors during the C++ compilation. The first would indicate that we cannot compare `query::first` to an integer and the second would pick the misspelling in `query::agee`. On the other hand, if we wrote something like this:

```
query q ("first = 123 AND agee = " + query::_ref (age));
```

It would compile fine and would trigger an error only when executed by the database system.

We can also combine the two query languages in a single query, for example:

```
query q ("first = 'John' AND" + (query::age < query::_ref (age)));
```

4.1 ODB Query Language

An ODB query is an expression that tells the database system whether any given object matches the desired criteria. As such, a query expression always evaluates as `true` or `false`. At the higher level, an expression consists of other expressions combined with logical operators such as `&&` (AND), `||` (OR), and `!` (NOT). For example:

```
using query = odb::query<person>;

query q (query::first == "John" || query::age == 31);
```

At the core of every query expression lie simple expressions which involve one or more object members, values, or parameters. To refer to an object member we use an expression such as `query::first` above. The names of members in the `query` class are derived from the names of data members in the object class by removing the common member name decorations, such as leading and trailing underscores, the `m_` prefix, etc.

In a simple expression an object member can be compared to a value, parameter, or another member using a number of predefined operators and functions. The following table gives an overview of the available expressions:

Operator	Description	Example
`==`	equal	`query::age == 31`
`!=`	unequal	`query::age != 31`
`<`	less than	`query::age < 31`
`>`	greater than	`query::age > 31`
`<=`	less than or equal	`query::age <= 31`
`>=`	greater than or equal	`query::age >= 31`
`in()`	one of the values	`query::age.in (30, 32, 34)`
`in_range()`	one of the values in range	`query::age.in_range (begin, end)`
`like()`	matches a pattern	`query::first.like ("J%")`
`is_null()`	value is `NULL`	`query::age.is_null ()`
`is_not_null()`	value is `NOT NULL`	`query::age.is_not_null ()`

The `in()` function accepts a maximum of five arguments. Use the `in_range()` function if you need to compare to more than five values. This function accepts a pair of standard C++ iterators and compares to all the values from the `begin` position inclusive and until and excluding the `end` position. The following code fragment shows how we can use these functions:

```
std::vector<string> names;

names.push_back ("John");
names.push_back ("Jack");
names.push_back ("Jane");

query q1 (query::first.in ("John", "Jack", "Jane"));
query q2 (query::first.in_range (names.begin (), names.end ()));
```

Note that the `like()` function does not perform any translation of the database system-specific extensions of the SQL `LIKE` operator. As a result, if you would like your application to be portable among various database systems, then limit the special characters used in the pattern to `%` (matches zero or more characters) and _ (matches exactly one character). It is also possible to specify the escape character as a second argument to the `like()` function. This character can then be used to escape the special characters (`%` and _) in the pattern. For example, the following query will match any two characters separated by an underscore:

```
query q (query::name.like ("_!__", "!"));
```

The operator precedence in the query expressions are the same as for equivalent C++ operators. We can use parentheses to make sure the expression is evaluated in the desired order. For example:

```
query q ((query::first == "John" || query::first == "Jane") &&
         query::age < 31);
```

4.2 Parameter Binding

An instance of the `odb::query` class encapsulates two parts of information about the query: the query expression and the query parameters. Parameters can be bound to C++ variables either by value or by reference.

If a parameter is bound by value, then the value for this parameter is copied from the C++ variable to the query instance at the query construction time. On the other hand, if a parameter is bound by reference, then the query instance stores a reference to the bound variable. The actual value of the parameter is only extracted at the query execution time. Consider, for example, the following two queries:

```
string name ("John");

query q1 (query::first == query::_val (name));
query q2 (query::first == query::_ref (name));

name = "Jane";

db.query<person> (q1); // Find John.
db.query<person> (q2); // Find Jane.
```

The `odb::query` class provides two special functions, `_val()` and `_ref()`, that allow us to bind the parameter either by value or by reference, respectively. In the ODB query language, if the binding is not specified explicitly, the value semantic is used by default. In the native query language, binding must always be specified explicitly. For example:

```
query q1 (query::age < age);                // By value.
query q2 (query::age < query::_val (age));  // By value.
query q3 (query::age < query::_ref (age));  // By reference.

query q4 ("age < " + age);                  // Error.
query q5 ("age < " + query::_val (age));    // By value.
query q6 ("age < " + query::_ref (age));    // By reference.
```

A query that only has by-value parameters does not depend on any other variables and is self-sufficient once constructed. A query that has one or more by-reference parameters depends on the bound variables until the query is executed. If one such variable goes out of scope and we execute the query, the behavior is undefined.

4.3 Executing a Query

Once we have the query instance ready and by-reference parameters initialized, we can execute the query using the `database::query()` function template. It has two overloaded versions:

```
template <typename T>
result<T>
query (bool cache = true);

template <typename T>
result<T>
query (const odb::query<T>&, bool cache = true);
```

The first `query()` function is used to return all the persistent objects of a given type stored in the database. The second function uses the passed query instance to only return objects matching the query criteria. The `cache` argument determines whether the objects' states should be cached in the application's memory or if they should be returned by the database system one by one as the iteration over the result progresses. The result caching is discussed in detail in the next section.

When calling the `query()` function, we have to explicitly specify the object type we are querying. For example:

```
using query = odb::query<person>;
using result = odb::result<person>;

result all (db.query<person> ());
result johns (db.query<person> (query::first == "John"));
```

Note that it is not required to explicitly create a named query variable before executing it. For example, the following two queries are equivalent:

```
query q (query::first == "John");

result r1 (db.query<person> (q));
result r1 (db.query<person> (query::first == "John"));
```

Normally, we would create a named query instance if we are planning to run the same query multiple times and would use the in-line version for those that are executed only once (see also Section 4.5, "Prepared Queries" for a more optimal way to re-execute the same query multiple times). A named query instance that does not have any by-reference parameters is immutable and

can be shared between multiple threads without synchronization. On the other hand, a query instance with by-reference parameters is modified every time it is executed. If such a query is shared among multiple threads, then access to this query instance must be synchronized from the execution point and until the completion of the iteration over the result.

It is also possible to create queries from other queries by combining them using logical operators. For example:

```
result
find_minors (database& db, const query& name_query)
{
  return db.query<person> (name_query && query::age < 18);
}

result r (find_minors (db, query::first == "John"));
```

The result of executing a query is zero, one, or more objects matching the query criteria. The `query()` function returns this result as an instance of the `odb::result` class template, which provides a stream-like interface and is discussed in detail in the next section.

In situations where we know that a query produces at most one element, we can instead use the `database::query_one()` and `database::query_value()` shortcut functions, for example:

```
  using query = odb::query<person>;

  unique_ptr<person> p (
    db.query_one<person> (
      query::email == "jon@example.com"));
```

The shortcut query functions have the following signatures:

```
  template <typename T>
  typename object_traits<T>::pointer_type
  query_one ();

  template <typename T>
  bool
  query_one (T&);

  template <typename T>
  T
  query_value ();

  template <typename T>
  typename object_traits<T>::pointer_type
  query_one (const odb::query<T>&);

  template <typename T>
```

```
bool
query_one (const odb::query<T>&, T&);

template <typename T>
T
query_value (const odb::query<T>&);
```

Similar to `query()`, the first three functions are used to return the only persistent object of a given type stored in the database. The second three versions use the passed query instance to only return the object matching the query criteria.

Similar to the `database::find()` functions (Section 3.9, "Loading Persistent Objects"), `query_one()` can either allocate a new instance of the object class in the dynamic memory or it can load the object's state into an existing instance. The `query_value()` function allocates and returns the object by value.

The `query_one()` function allows us to determine if the query result contains zero or one element. If no objects matching the query criteria were found in the database, the first version of `query_one()` returns the `NULL` pointer while the second — `false`. If the second version returns `false`, then the passed object remains unchanged. For example:

```
if (unique_ptr<person> p = db.query_one<person> (
      query::email == "jon@example.com"))
{
  ...
}

person p;
if (db.query_one<person> (query::email == "jon@example.com", p))
{
  ...
}
```

If the query executed using `query_one()` or `query_value()` returns more than one element, then these functions fail with an assertion. Additionally, `query_value()` also fails with an assertion if the query returned no elements.

Common situations where we can use the shortcut functions are a query condition that uses a data member with the `unique` constraint (at most one element returned; see Section 14.7, "Index Definition Pragmas") as well as aggregate queries (exactly one element returned; see Chapter 10, "Views").

4.4 Query Result

The `database::query()` function returns the result of executing a query as an instance of the `odb::result` class template, for example:

```
using query = odb::query<person>;
using result = odb::result<person>;

result johns (db.query<person> (query::first == "John"));
```

It is best to view an instance of `odb::result` as a handle to a stream, such as a socket stream. While we can make a copy of a result or assign one result to another, the two instances will refer to the same result stream. Advancing the current position in one instance will also advance it in another. The result instance is only usable within the transaction it was created in. Trying to manipulate the result after the transaction has terminated leads to undefined behavior.

The `odb::result` class template conforms to the standard C++ sequence requirements and has the following interface:

```
namespace odb
{
  template <typename T>
  class result
  {
  public:
    using iterator = odb::result_iterator<T>;

  public:
    result ();

    result (const result&);

    result&
    operator= (const result&);

    void
    swap (result&)

  public:
    iterator
    begin ();

    iterator
    end ();

  public:
    void
    cache ();
```

```
    bool
    empty () const;

    std::size_t
    size () const;
  };
}
```

The default constructor creates an empty result set. The `cache()` function caches the returned objects' state in the application's memory. We have already mentioned result caching when we talked about query execution. As you may remember the `database::query()` function caches the result unless instructed not to by the caller. The `cache()` function allows us to cache the result at a later stage if it wasn't already cached during query execution.

If the result is cached, the database state of all the returned objects is stored in the application's memory. Note that the actual objects are still only instantiated on demand during result iteration. It is the raw database state that is cached in memory. In contrast, for uncached results the object's state is sent by the database system one object at a time as the iteration progresses.

Uncached results can improve the performance of both the application and the database system in situations where we have a large number of objects in the result or if we will only examine a small portion of the returned objects. However, uncached results have a number of limitations. There can only be one uncached result in a transaction. Creating another result (cached or uncached) by calling `database::query()` will invalidate the existing uncached result. Furthermore, calling any other database functions, such as `update()` or `erase()` will also invalidate the uncached result. It also follows that uncached results cannot be used on objects with containers (Chapter 5, "Containers") since loading a container would invalidate the uncached result.

The `empty()` function returns `true` if there are no objects in the result and `false` otherwise. The `size()` function can only be called for cached results. It returns the number of objects in the result. If we call this function on an uncached result, the `odb::result_not_cached` exception is thrown.

To iterate over the objects in a result we use the `begin()` and `end()` functions together with the `odb::result<T>::iterator` type, for example:

```
result r (db.query<person> (query::first == "John"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  ...
}
```

In C++11 we can use the `auto`-typed variabe instead of spelling the iterator type explicitly, for example:

```
for (auto i (r.begin ()); i != r.end (); ++i)
{
  ...
}
```

The C++11 range-based `for`-loop can be used to further simplify the iteration:

```
for (person& p: r)
{
  ...
}
```

The result iterator is an input iterator which means that the only two position operations that it supports are to move to the next object and to determine whether the end of the result stream has been reached. In fact, the result iterator can only be in two states: the current position and the end position. If we have two iterators pointing to the current position and then we advance one of them, the other will advance as well. This, for example, means that it doesn't make sense to store an iterator that points to some object of interest in the result stream with the intent of dereferencing it after the iteration is over. Instead, we would need to store the object itself. We also cannot iterate over the same result multiple times without re-executing the query.

The result iterator has the following dereference functions that can be used to access the pointed-to object:

```
namespace odb
{
  template <typename T>
  class result_iterator
  {
  public:
    T*
    operator-> () const;

    T&
    operator* () const;

    typename object_traits<T>::pointer_type
    load ();

    void
    load (T& x);

    typename object_traits<T>::id_type
    id ();
  };
}
```

When we call the `*` or `->` operator, the iterator will allocate a new instance of the object class in the dynamic memory, load its state from the database state, and return a reference or pointer to the new instance. The iterator maintains the ownership of the returned object and will return the same pointer for subsequent calls to either of these operators until it is advanced to the next object or we call the first `load()` function (see below). For example:

```
result r (db.query<person> (query::first == "John"));

for (result::iterator i (r.begin ()); i != r.end ();)
{
  cout << i->last () << endl; // Create an object.
  person& p (*i);             // Reference to the same object.
  cout << p.age () << endl;
  ++i;                        // Free the object.
}
```

The overloaded `result_iterator::load()` functions are similar to `database::load()`. The first function returns a dynamically allocated instance of the current object. As an optimization, if the iterator already owns an object as a result of an earlier call to the `*` or `->` operator, then it relinquishes the ownership of this object and returns it instead. This allows us to write code like this without worrying about a double allocation:

```
result r (db.query<person> (query::first == "John"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  if (i->last == "Doe")
  {
    unique_ptr p (i.load ());
    ...
  }
}
```

Note, however, that because of this optimization, a subsequent to `load()` call to the `*` or `->` operator results in the allocation of a new object.

The second `load()` function allows us to load the current object's state into an existing instance. For example:

```
result r (db.query<person> (query::first == "John"));

person p;
for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  i.load (p);
  cout << p.last () << endl;
  cout << i.age () << endl;
}
```

The `id()` function return the object id of the current object. While we can achieve the same by loading the object and getting its id, this function is more efficient since it doesn't actually create the object. This can be useful when all we need is the object's identifier. For example:

```
std::set<unsigned long long> set = ...; // Persons of interest.

result r (db.query<person> (query::first == "John"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  if (set.find (i.id ()) != set.end ()) // No object loaded.
  {
    cout << i->first () << endl; // Object loaded.
  }
}
```

4.5 Prepared Queries

Most modern relational database systems have the notion of a prepared statement. Prepared statements allow us to perform the potentially expensive tasks of parsing SQL, preparing the query execution plan, etc., once and then executing the same query multiple times, potentially using different values for parameters in each execution.

In ODB all the non-query database operations such as `persist()`, `load()`, `update()`, etc., are implemented in terms of prepared statements that are cached and reused. While the `query()`, `query_one()`, and `query_value()` database operations also use prepared statements, these statements are not cached or reused by default since ODB has no knowledge of whether a query will be executed multiple times or only once. Instead, ODB provides a mechanism, called prepared queries, that allows us to prepare a query once and execute it multiple times. In other words, ODB prepared queries are a thin wrapper around the underlying database's prepared statement functionality.

In most cases ODB shields the application developer from database connection management and multi-threading issues. However, when it comes to prepared queries, a basic understanding of how ODB manages these aspects is required. Conceptually, the `odb::database` class represents a specific database, that is, a data store. However, underneath, it maintains one or more connections to this database. A connection can be used only by a single thread at a time. When we start a transaction (by calling `database::begin()`), the transaction instance obtains a connection and holds on to it until the transaction is committed or rolled back. During this time no other thread can use this connection. When the transaction releases the connection, it may be closed or reused by another transaction in this or another thread. What exactly happens to a connection after it has been released depends on the connection factory that is used by the `odb::database` instance. For more information on connection factories, refer to Part II, "Database Systems".

A query prepared on one connection cannot be executed on another. In other words, a prepared query is associated with the connection. One important implication of this restriction is that we cannot prepare a query in one transaction and then try to execute it in another without making sure that both transactions use the same connection.

To enable the prepared query functionality we need to specify the `--generate-prepared` ODB compiler option. If we are planning to always prepare our queries, then we can disable the once-off query execution support by also specifying the `--omit-unprepared` option.

To prepare a query we use the `prepare_query()` function template. This function can be called on both the `odb::database` and `odb::connection` instances. The `odb::database` version simply obtains the connection used by the currently active transaction and calls the corresponding `odb::connection` version. If no transaction is currently active, then this function throws the `odb::not_in_transaction` exception (Section 3.5, "Transactions"). The `prepare_query()` function has the following signature:

```
template <typename T>
prepared_query<T>
prepare_query (const char* name, const odb::query<T>&);
```

The first argument to the `prepare_query()` function is the prepared query name. This name is used as a key for prepared query caching (discussed later) and must be unique. For some databases, notably PostgreSQL, it is also used as a name of the underlying prepared statement. The name "*object*`_query`" (for example, "`person_query`") is reserved for the once-off queries executed by the `database::query()` function. Note that the `prepare_query()` function makes only a shallow copy of this argument, which means that the name must be valid for the lifetime of the returned `prepared_query` instance.

The second argument to the `prepare_query()` function is the query criteria. It has the same semantics as in the `query()` function discussed in Section 4.3, "Executing a Query". Similar to `query()`, we also have to explicitly specify the object type that we will be querying. For example:

```
using query = odb::query<person>;
using prep_query = odb::prepared_query<person>;

prep_query pq (
  db.prepare_query<person> ("person-age-query", query::age > 50));
```

The result of executing the `prepare_query()` function is the `prepared_query` instance that represent the prepared query. It is best to view `prepared_query` as a handle to the underlying prepared statement. While we can make a copy of it or assign one `prepared_query` to another, the two instances will refer to the same prepared statement. Once the last instance of `prepared_query` referencing a specific prepared statement is destroyed, this statement is released. The `prepared_query` class template has the following interface:

```
namespace odb
{
  template <typename T>
  struct prepared_query
  {
    prepared_query ();

    prepared_query (const prepared_query&)
    prepared_query& operator= (const prepared_query&)

    result<T>
    execute (bool cache = true);

    typename object_traits<T>::pointer_type
    execute_one ();

    bool
    execute_one (T& object);

    T
    execute_value ();

    const char*
    name () const;

    statement&
    statement () const;

    operator unspecified_bool_type () const;
  };
}
```

The default constructor creates an empty `prepared_query` instance, that is, an instance that does not reference a prepared statement and therefore cannot be executed. The only way to create a non-empty prepared query is by calling the `prepare_query()` function discussed above. To test whether the prepared query is empty, we can use the implicit conversion operator to a boolean type. For example:

```
prepared_query<person> pq;

if (pq)
{
  // Not empty.
  ...
}
```

The `execute()` function executes the query and returns the result instance. The `cache` argument indicates whether the result should be cached and has the same semantics as in the `query()` function. In fact, conceptually, `prepare_query()` and `execute()` are just the

`query()` function split into two: `prepare_query()` takes the first `query()` argument (the query condition) while `execute()` takes the second (the cache flag). Note also that re-executing a prepared query invalidates the previous execution result, whether cached or uncached.

The `execute_one()` and `execute_value()` functions can be used as shortcuts to execute a query that is known to return at most one or exactly one object, respectively. The arguments and return values in these functions have the same semantics as in `query_one()` and `query_value()`. And similar to `execute()` above, `prepare_query()` and `execute_one/value()` can be seen as the `query_one/value()` function split into two: `prepare_query()` takes the first `query_one/value()` argument (the query condition) while `execute_one/value()` takes the second argument (if any) and returns the result. Note also that `execute_one/value()` never caches its result but invalidates the result of any previous `execute()` call on the same prepared query.

The `name()` function returns the prepared query name. This is the same name as was passed as the first argument in the `prepare_query()` call. The `statement()` function returns a reference to the underlying prepared statement. Note also that calling any of these functions on an empty `prepared_query` instance results in undefined behavior.

The simplest use-case for a prepared query is the need to execute the same query multiple times within a single transaction. Consider the following example that queries for people that are older than a number of different ages. This and subsequent code fragments are taken from the `prepared` example in the `odb-examples` package.

```
using query = odb::query<person>;
using prep_query = odb::prepared_query<person>;
using result = odb::result<person>;

transaction t (db.begin ());

unsigned short age;
query q (query::age > query::_ref (age));
prep_query pq (db.prepare_query<person> ("person-age-query", q));

for (age = 90; age > 40; age -= 10)
{
  result r (pq.execute ());
  ...
}

t.commit ();
```

Another scenario is the need to reuse the same query in multiple transactions that are executed at once. As was mentioned above, in this case we need to make sure that the prepared query and all the transactions use the same connection. Consider an alternative version of the above example that executes each query in a separate transaction:

```
connection_ptr conn (db.connection ());

unsigned short age;
query q (query::age > query::_ref (age));
prep_query pq (conn->prepare_query<person> ("person-age-query", q));

for (age = 90; age > 40; age -= 10)
{
  transaction t (conn->begin ());

  result r (pq.execute ());
  ...

  t.commit ();
}
```

Note that with this approach we hold on to the database connection until all the transactions involving the prepared query are executed. In particular, this means that while we are busy, the connection cannot be reused by another thread. Therefore, this approach is only recommended if all the transactions are executed close to each other. Also note that an uncached (see below) prepared query is invalidated once we release the connection on which it was prepared.

If we need to reuse a prepared query in transactions that are executed at various times, potentially in different threads, then the recommended approach is to cache the prepared query on the connection. To support this functionality the `odb::database` and `odb::connection` classes provide the following function templates. Similar to `prepare_query()`, the `odb::database` versions of the below functions call the corresponding `odb::connection` versions using the currently active transaction to resolve the connection.

```
template <typename T>
void
cache_query (const prepared_query<T>&);

template <typename T, typename P>
void
cache_query (const prepared_query<T>&,
             std::[auto|unique]_ptr<P> params);

template <typename T>
prepared_query<T>
lookup_query (const char* name);

template <typename T, typename P>
prepared_query<T>
lookup_query (const char* name, P*& params);
```

The `cache_query()` function caches the passed prepared query on the connection. The second overloaded version of `cache_query()` also takes a pointer to the by-reference query parameters. In C++98/03 it should be `std::auto_ptr` while in C++11 — `std::unique_ptr`. The `cache_query()` function assumes ownership of the passed `params` argument. If a prepared query with the same name is already cached on this connection, then the `odb::prepared_already_cached` exception is thrown.

The `lookup_query()` function looks up a previously cached prepared query given its name. The second overloaded version of `lookup_query()` also returns a pointer to the by-reference query parameters. If a prepared query with this name has not been cached, then an empty `prepared_query` instance is returned. If a prepared query with this name has been cached but either the object type or the parameters type does not match that which was cached, then the `odb::prepared_type_mismatch` exception is thrown.

As a first example of the prepared query cache functionality, consider the case that does not use any by-reference parameters:

```
for (unsigned short i (0); i < 5; ++i)
{
  transaction t (db.begin ());

  prep_query pq (db.lookup_query<person> ("person-age-query"));

  if (!pq)
  {
    pq = db.prepare_query<person> (
      "person-val-age-query", query::age > 50);
    db.cache_query (pq);
  }

  result r (pq.execute ());
  ...

  t.commit ();

  // Do some other work.
  //
  ...
}
```

The following example shows how to do the same but for a query that includes by-reference parameters. In this case the parameters are cached together with the prepared query.

```
for (unsigned short age (90); age > 40; age -= 10)
{
  transaction t (db.begin ());

  unsigned short* age_param;
```

```
  prep_query pq (
    db.lookup_query<person> ("person-age-query", age_param));

  if (!pq)
  {
    unique_ptr<unsigned short> p (new unsigned short);
    age_param = p.get ();
    query q (query::age > query::_ref (*age_param));
    pq = db.prepare_query<person> ("person-age-query", q);
    db.cache_query (pq, p); // Assumes ownership of p.
  }

  *age_param = age; // Initialize the parameter.
  result r (pq.execute ());
  ...

  t.commit ();

  // Do some other work.
  //
  ...
}
```

As is evident from the above examples, when we use a prepared query cache, each transaction that executes a query must also include code that prepares and caches this query if it hasn't already been done. If a prepared query is used in a single place in the application, then this is normally not an issue since all the relevant code is kept in one place. However, if the same query is used in several different places in the application, then we may end up duplicating the same preparation and caching code, which makes it hard to maintain.

To resolve this issue ODB allows us to register a prepared query factory that will be called to prepare and cache a query during the call to `lookup_query()`. To register a factory we use the `database::query_factory()` function. In C++98/03 it has the following signature:

```
  void
  query_factory (const char* name,
                 void (*factory) (const char* name, connection&));
```

While in C++11 it uses the `std::function` class template:

```
  void
  query_factory (const char* name,
                 std::function<void (const char* name, connection&)>);
```

The first argument to the `query_factory()` function is the prepared query name that this factory will be called to prepare and cache. An empty name is treated as a fallback wildcard factory that is capable of preparing any query. The second argument is the factory function or, in C++11, function object or lambda.

The example fragment shows how we can use the prepared query factory:

```
struct params
{
  unsigned short age;
  string first;
};

static void
query_factory (const char* name, connection& c)
{
  unique_ptr<params> p (new params);
  query q (query::age > query::_ref (p->age) &&
           query::first == query::_ref (p->first));
  prep_query pq (c.prepare_query<person> (name, q));
  c.cache_query (pq, p);
}

db.query_factory ("person-age-name-query", &query_factory);

for (unsigned short age (90); age > 40; age -= 10)
{
  transaction t (db.begin ());

  params* p;
  prep_query pq (db.lookup_query<person> ("person-age-name-query", p));
  assert (pq);

  p->age = age;
  p->first = "John";
  result r (pq.execute ());
  ...

  t.commit ();
}
```

Instead of a static function we could have used a lambda:

```
db.query_factory (
  "person-age-name-query",
  [] (const char* name, connection& c)
  {
    unique_ptr<params> p (new params);
    query q (query::age > query::_ref (p->age) &&
             query::first == query::_ref (p->first));
    prep_query pq (c.prepare_query<person> (name, q));
    c.cache_query (pq, std::move (p));
  });
```

Note that the `database::query_factory()` function is not thread-safe and should be called before starting any threads that may require this functionality. Normally, all the prepared

query factories are registered as part of the database instance creation.

5 Containers

The ODB runtime library provides built-in persistence support for all the commonly used standard C++98/03 containers, namely, `std::vector`, `std::list`, `std::deque`, `std::set`, `std::multiset`, `std::map`, and `std::multimap` as well as C++11 `std::array`, `std::forward_list`, `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, and `std::unordered_multimap`. Plus, ODB profile libraries, that are available for commonly used frameworks and libraries (such as Boost and Qt), provide persistence support for containers found in these frameworks and libraries (Part III, "Profiles"). Both the ODB runtime library and profile libraries also provide a number of change-tracking container equivalents which can be used to minimize the number of database operations necessary to synchronize the container state with the database (Section 5.4, "Change-Tracking Containers"). It is also easy to persist custom container types as discussed later in Section 5.5, "Using Custom Containers".

We don’t need to do anything special to declare a member of a container type in a persistent class. For example:

```
#pragma db object
class person
{
  ...
private:
  std::vector<std::string> nicknames_;
  ...
};
```

The complete version of the above code fragment and the other code samples presented in this chapter can be found in the `container` example in the `odb-examples` package.

A data member in a persistent class that is of a container type behaves like a value type. That is, when an object is made persistent, the elements of the container are stored in the database. Similarly, when a persistent object is loaded from the database, the contents of the container are automatically loaded as well. A data member of a container type can also use a smart pointer, as discussed in Section 7.3, "Pointers and `NULL` Value Semantics".

While an ordinary member is mapped to one or more columns in the object’s table, a member of a container type is mapped to a separate table. The exact schema of such a table depends on the kind of container. ODB defines the following container kinds: ordered, set, multiset, map, and multimap. The container kinds and the contents of the tables to which they are mapped are discussed in detail in the following sections.

Containers in ODB can contain simple value types (Section 7.1, "Simple Value Types"), composite value types (Section 7.2, "Composite Value Types"), and pointers to objects (Chapter 6, "Relationships"). Containers of containers, either directly or indirectly via a composite value type, are not allowed. A key in a map or multimap container can be a simple or composite value type but not a pointer to an object. An index in the ordered container should be a simple integer value type.

The value type in the ordered, set, and map containers as well as the key type in the map containers should be default-constructible. The default constructor in these types can be made private in which case the `odb::access` class should be made a friend of the value or key type. For example:

```
#pragma db value
class name
{
public:
  name (const std::string&, const std::string&);
  ...
private:
  friend class odb::access;
  name ();
  ...
};

#pragma db object
class person
{
  ...
private:
  std::vector<name> aliases_;
  ...
};
```

5.1 Ordered Containers

In ODB an ordered container is any container that maintains (explicitly or implicitly) an order of its elements in the form of an integer index. Standard C++ containers that are ordered include `std::vector` `std::list`, and `std::deque` as well as C++11 `std::array` and `std::forward_list`. While elements in `std::set` are also kept in a specific order, this order is not based on an integer index but rather on the relationship between elements. As a result, `std::set` is not considered an ordered container for the purpose of persistence.

The database table for an ordered container consists of at least three columns. The first column contains the object id of a persistent class instance of which the container is a member. The second column contains the element index within a container. And the last column contains the element value. If the object id or element value are composite, then, instead of a single column,

they can occupy multiple columns. For an ordered container table the ODB compiler also defines two indexes: one for the object id column(s) and the other for the index column. Refer to Section 14.7, "Index Definition Pragmas" for more information on how to customize these indexes.

Consider the following persistent object as an example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id auto
  unsigned long long id_;

  std::vector<std::string> nicknames_;
  ...
};
```

The resulting database table (called `person_nicknames`) will contain the object id column of type `unsigned long` (called `object_id`), the index column of an integer type (called `index`), and the value column of type `std::string` (called `value`).

A number of ODB pragmas allow us to customize the table name, column names, and native database types of an ordered container both, on the per-container and per-member basis. For more information on these pragmas, refer to Chapter 14, "ODB Pragma Language". The following example shows some of the possible customizations:

```
#pragma db object
class person
{
  ...
private:
  #pragma db table("nicknames")               \
              id_column("person_id")          \
              index_type("SMALLINT UNSIGNED") \
              index_column("nickname_number") \
              value_type("VARCHAR(255)")      \
              value_column("nickname")
  std::vector<std::string> nicknames_;
  ...
};
```

While the C++ container used in a persistent class may be ordered, sometimes we may wish to store such a container in the database without the order information. In the example above, for instance, the order of person's nicknames is probably not important. To instruct the ODB compiler to ignore the order in ordered containers we can use the `db unordered` pragma (Section 14.3.9, "`unordered`", Section 14.4.19, "`unordered`"). For example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db unordered
  std::vector<std::string> nicknames_;
  ...
};
```

The table for an ordered container that is marked unordered won't have the index column and the order in which elements are retrieved from the database may not be the same as the order in which they were stored.

5.2 Set and Multiset Containers

In ODB set and multiset containers (referred to as just set containers) are associative containers that contain elements based on some relationship between them. A set container may or may not guarantee a particular order of the elements that it stores. Standard C++ containers that are considered set containers for the purpose of persistence include `std::set` and `std::multiset` as well as C++11 `std::unordered_set` and `std::unordered_multiset`.

The database table for a set container consists of at least two columns. The first column contains the object id of a persistent class instance of which the container is a member. And the second column contains the element value. If the object id or element value are composite, then, instead of a single column, they can occupy multiple columns. ODB compiler also defines an index on a set container table for the object id column(s). Refer to Section 14.7, "Index Definition Pragmas" for more information on how to customize this index.

Consider the following persistent object as an example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id auto
  unsigned long long id_;

  std::set<std::string> emails_;
  ...
};
```

The resulting database table (called `person_emails`) will contain the object id column of type `unsigned long` (called `object_id`) and the value column of type `std::string` (called `value`).

A number of ODB pragmas allow us to customize the table name, column names, and native database types of a set container, both on the per-container and per-member basis. For more information on these pragmas, refer to Chapter 14, "ODB Pragma Language". The following example shows some of the possible customizations:

```
#pragma db object
class person
{
  ...
private:
  #pragma db table("emails")             \
             id_column("person_id")      \
             value_type("VARCHAR(255)") \
             value_column("email")
  std::set<std::string> emails_;
  ...
};
```

5.3 Map and Multimap Containers

In ODB map and multimap containers (referred to as just map containers) are associative containers that contain key-value elements based on some relationship between keys. A map container may or may not guarantee a particular order of the elements that it stores. Standard C++ containers that are considered map containers for the purpose of persistence include `std::map` and `std::multimap` as well as C++11 `std::unordered_map` and `std::unordered_multimap`.

The database table for a map container consists of at least three columns. The first column contains the object id of a persistent class instance of which the container is a member. The second column contains the element key. And the last column contains the element value. If the object id, element key, or element value are composite, then instead of a single column they can occupy multiple columns. ODB compiler also defines an index on a map container table for the object id column(s). Refer to Section 14.7, "Index Definition Pragmas" for more information on how to customize this index.

Consider the following persistent object as an example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id auto
  unsigned long long id_;
```

```
  std::map<unsigned short, float> age_weight_map_;
  ...
};
```

The resulting database table (called `person_age_weight_map`) will contain the object id column of type `unsigned long` (called `object_id`), the key column of type `unsigned short` (called `key`), and the value column of type `float` (called `value`).

A number of ODB pragmas allow us to customize the table name, column names, and native database types of a map container, both on the per-container and per-member basis. For more information on these pragmas, refer to Chapter 14, "ODB Pragma Language". The following example shows some of the possible customizations:

```
#pragma db object
class person
{
  ...
private:
  #pragma db table("weight_map")       \
              id_column("person_id")   \
              key_type("INT UNSIGNED") \
              key_column("age")        \
              value_type("DOUBLE")     \
              value_column("weight")
  std::map<unsigned short, float> age_weight_map_;
  ...
};
```

5.4 Change-Tracking Containers

When a persistent object containing one of the standard containers is updated in the database, ODB has no knowledge of which elements were inserted, erased, or modified. As a result, ODB has no choice but to assume the whole container has changed and update the state of every single element. This can result in a significant overhead if a container contains a large number of elements and we only changed a small subset of them.

To eliminate this overhead, ODB provides a notion of *change-tracking containers*. A change-tracking container, besides containing its elements, just like an ordinary container, also includes the change state for each element. When it is time to update such a container in the database, ODB can use this change information to perform a minimum number of database operations necessary to synchronize the container state with the database.

The current version of the ODB runtime library provides a change-tracking equivalent of `std::vector` (Section 5.4.1, "Change-Tracking `vector`") with support for other standard container equivalents planned for future releases. ODB profile libraries also provide

change-tracking equivalents for some containers found in the corresponding frameworks and libraries (Part III, "Profiles").

A change-tracking container equivalent can normally be used as a drop-in replacement for an ordinary container except for a few minor interface differences (discussed in the corresponding sub-sections). In particular, we don't need to do anything extra to effect change tracking. ODB will automatically start, stop, and reset change tracking when necessary. The following example illustrates this point using `odb::vector` as a replacement for `std::vector`.

```
#pragma db object
class person
{
  ...

  odb::vector<std::string> names;
};

person p; // No change tracking (not persistent).
p.names.push_back ("John Doe");

{
  transaction t (db.begin ());
  db.persist (p); // Start change tracking (persistent).
  t.commit ();
}

p.names.push_back ("Johnny Doo");

{
  transaction t (db.begin ());
  db.update (p); // One INSERT; reset change state.
  t.commit ();
}

p.names.modify (0) = "Doe, John"; // Instead of operator[].
p.names.pop_back ();

{
  transaction t (db.begin ());
  db.update (p); // One UPDATE, one DELETE; reset change state.
  t.commit ();
}

{
  transaction t (db.begin ());
  unique_ptr<person> p1 (db.load<person> (...)); // Start change tracking.
  p1->names.insert (p1->names.begin (), "Joe Do");
  db.update (*p1); // One UPDATE, one INSERT; reset change state.
  t.commit ();
}
```

```
{
  transaction t (db.begin ());
  db.erase (p); // One DELETE; stop change tracking (not persistent).
  t.commit ();
}
```

One interesting aspect of change tracking is what happens when a transaction that contains an update is later rolled back. In this case, while the change-tracking container has reset the change state (after update), actual changes were not committed to the database. Change-tracking containers handle this case by automatically registering a rollback callback and then, if it is called, marking the container as "completely changed". In this state, the container no longer tracks individual element changes and, when updated, falls back to the complete state update, just like an ordinary container. The following example illustrates this point:

```
person p;
p.names.push_back ("John Doe");

{
  transaction t (db.begin ());
  db.persist (p); // Start change tracking (persistent).
  t.commit ();
}

p.names.push_back ("Johnny Doo");

for (;;)
{
  try
  {
    transaction t (db.begin ());

    // First try: one INSERT.
    // Next try: one DELETE, two INSERTs.
    //
    db.update (p); // Reset change state.

    t.commit (); // If throws (rollback), mark as completely changed.
    break;
  }
  catch (const odb::recoverable&)
  {
    continue;
  }
}
```

For the interaction of change-tracking containers with change-updated object sections, refer to Section 9.4, "Sections and Change-Tracking Containers". Note also that change-tracking containers cannot be accessed with by-value accessors (Section 14.4.5, "`get`/`set`/`access`") since in

certain situations such access may involve a modification of the container (for example, clearing the change flag after update).

5.4.1 Change-Tracking `vector`

Class template `odb::vector`, defined in `<odb/vector.hxx>`, is a change-tracking equivalent for `std::vector`. It is implemented in terms of `std::vector` and is implicit-convertible to and implicit-constructible from `const std::vector&`. In particular, this means that we can use `odb::vector` instance anywhere `const std::vector&` is expected. In addition, `odb::vector` constant iterator (`const_iterator`) is the same type as that of `std::vector`.

`odb::vector` incurs 2-bit per element overhead in order to store the change state. It cannot be stored unordered in the database (Section 14.4.19 "`unordered`") but can be used as an inverse side of a relationship (6.2 "Bidirectional Relationships"). In this case, no change tracking is performed since no state for such a container is stored in the database.

The number of database operations required to update the state of `odb::vector` corresponds well to the complexity of `std::vector` functions. In particular, adding or removing an element from the back of the vector (for example, with `push_back()` and `pop_back()`), requires only a single database statement execution. In contrast, inserting or erasing an element somewhere in the middle of the vector will require a database statement for every element that follows it.

`odb::vector` replicates most of the `std::vector` interface as defined in both C++98/03 and C++11 standards. However, functions and operators that provide direct write access to the elements had to be altered or disabled in order to support change tracking. Additional functions used to interface with `std::vector` and to control the change tracking state were also added. The following listing summarizes the differences between the `odb::vector` and `std::vector` interfaces. Any `std::vector` function or operator not mentioned in this listing has exactly the same signature and semantics in `odb::vector`. Functions and operators that were disabled are shown as commented out and are followed by functions/operators that replace them.

```
namespace odb
{
  template <class T, class A = std::allocator<T>>
  class vector
  {
    ...

    // Element access.
    //

    //reference operator[] (size_type);
```

```
  reference modify (size_type);

//reference at (size_type);
  reference modify_at (size_type);

//reference front ();
  reference modify_front ();

//reference back ();
  reference modify_back ();

//T*        data () noexcept;
  T*        modify_data () noexcept; // C++11 only.

// Iterators.
//
using const_iterator = typename std::vector<T, A>::const_iterator;

class iterator
{
  ...

  // Element Access.
  //

  //reference       operator* () const;
    const_reference operator* () const;
    reference       modify () const;

  //pointer       operator-> () const;
    const_pointer operator-> () const;

  //reference       operator[] (difference_type);
    const_reference operator[] (difference_type);
    reference       modify (difference_type) const;

  // Interfacing with std::vector::iterator.
  //
  typename std::vector<T, A>::iterator base () const;
};

// Return std::vector iterators. The begin() functions mark
// all the elements as modified.
//
typename std::vector<T, A>::iterator         mbegin ();
typename std::vector<T, A>::iterator         mend ();
typename std::vector<T, A>::reverse_iterator mrbegin ();
typename std::vector<T, A>::reverse_iterator mrend ();

// Interfacing with std::vector.
//
```

```
    vector (const std::vector<T, A>&);
    vector (std::vector<T, A>&&); // C++11 only.

    vector& operator= (const std::vector<T, A>&);
    vector& operator= (std::vector<T, A>&&); // C++11 only.

    operator const std::vector<T, A>& () const;
    std::vector<T, A>& base ();
    const std::vector<T, A>& base ();

    // Change tracking.
    //
    bool _tracking () const;
    void _start () const;
    void _stop () const;
    void _arm (transaction&) const;
  };
}
```

The following example highlights some of the differences between the two interfaces. `std::vector` versions are commented out.

```
#include <vector>
#include <odb/vector.hxx>

void f (const std::vector<int>&);

odb::vector<int> v ({1, 2, 3});

f (v); // Ok, implicit conversion.

if (v[1] == 2) // Ok, const access.
  //v[1]++;
  v.modify (1)++;

//v.back () = 4;
v.modify_back () = 4;

for (auto i (v.begin ()); i != v.end (); ++i)
{
  if (*i != 0) // Ok, const access.
    //*i += 10;
    i.modify () += 10;
}

std::sort (v.mbegin (), v.mend ());
```

Note also the subtle difference between copy/move construction and copy/move assignment of `odb::vector` instances. While copy/move constructor will copy/move both the elements as well as their change state, in contrast, assignment is tracked as any other change to the vector

content.

5.5 Using Custom Containers

While the ODB runtime and profile libraries provide support for a wide range of containers, it is also easy to persist custom container types or make a change-tracking version out of one.

To achieve this you will need to implement the `container_traits` class template specialization for your container. First, determine the container kind (ordered, set, multiset, map, or multimap) for your container type. Then use a specialization for one of the standard C++ containers found in the common ODB runtime library (`libodb`) as a base for your own implementation.

Once the container traits specialization is ready for your container, you will need to include it into the ODB compilation process using the `--odb-epilogue` option and into the generated header files with the `--hxx-prologue` option. As an example, suppose we have a hash table container for which we have the traits specialization implemented in the `hashtable-traits.hxx` file. Then, we can create an ODB compiler options file for this container and save it to `hashtable.options`:

```
# Options file for the hash table container.
#
--odb-epilogue '#include "hashtable-traits.hxx"'
--hxx-prologue '#include "hashtable-traits.hxx"'
```

Now, whenever we compile a header file that uses the hashtable container, we can specify the following command line option to make sure it is recognized by the ODB compiler as a container and the traits file is included in the generated code:

```
--options-file hashtable.options
```

6 Relationships

Relationships between persistent objects are expressed with pointers or containers of pointers. The ODB runtime library provides built-in support for `std::shared_ptr`/`std::weak_ptr` (C++11), `std::unique_ptr` (C++11), `std::auto_ptr` (C++98/03 only), and raw pointers. Plus, ODB profile libraries, that are available for commonly used frameworks and libraries (such as Boost and Qt), provide support for smart pointers found in these frameworks and libraries (Part III, "Profiles"). It is also easy to add support for a custom smart pointer as discussed later in Section 6.5, "Using Custom Smart Pointers". Any supported smart pointer can be used in a data member as long as it can be explicitly constructed from the canonical object pointer (Section 3.3, "Object and View Pointers"). For example, we can use `weak_ptr` if the object pointer is `shared_ptr`.

When an object containing a pointer to another object is loaded, the pointed-to object is loaded as well. In some situations this eager loading of the relationships is undesirable since it can lead to a large number of otherwise unused objects being instantiated from the database. To support finer control over relationships loading, the ODB runtime and profile libraries provide the so-called *lazy* versions of the supported pointers. An object pointed-to by a lazy pointer is not loaded automatically when the containing object is loaded. Instead, we have to explicitly request the instantiation of the pointed-to object. Lazy pointers are discussed in detail in Section 6.4, "Lazy Pointers".

As a simple example, consider the following employee-employer relationship. Code examples presented in this chapter will use the C++11 `shared_ptr` and `weak_ptr` smart pointers from the `std` namespace.

```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  std::string first_name_;
```

```
  std::string last_name_;

  shared_ptr<employer> employer_;
};
```

By default, an object pointer can be `NULL`. To specify that a pointer always points to a valid object we can use the `not_null` pragma (Section 14.4.6, "`null`/`not_null`") for single object pointers and the `value_not_null` pragma (Section 14.4.28, "`value_null`/`value_not_null`") for containers of object pointers. For example:

```
#pragma db object
class employee
{
  ...

  #pragma db not_null
  shared_ptr<employer> current_employer_;

  #pragma db value_not_null
  std::vector<shared_ptr<employer>> previous_employers_;
};
```

In this case, if we call either `persist()` or `update()` database function on the `employee` object and the `current_employer_` pointer or one of the pointers stored in the `previous_employers_` container is `NULL`, then the `odb::null_pointer` exception will be thrown.

We don't need to do anything special to establish or navigate a relationship between two persistent objects, as shown in the following code fragment:

```
// Create an employer and a few employees.
//
unsigned long long john_id, jane_id;
{
  shared_ptr<employer> er (new employer ("Example Inc"));
  shared_ptr<employee> john (new employee ("John", "Doe"));
  shared_ptr<employee> jane (new employee ("Jane", "Doe"));

  john->employer_ = er;
  jane->employer_ = er;

  transaction t (db.begin ());

  db.persist (er);
  john_id = db.persist (john);
  jane_id = db.persist (jane);

  t.commit ();
}
```

```
// Load a few employee objects and print their employer.
//
{
  session s;
  transaction t (db.begin ());

  shared_ptr<employee> john (db.load<employee> (john_id));
  shared_ptr<employee> jane (db.load<employee> (jane_id));

  cout << john->employer_->name_ << endl;
  cout << jane->employer_->name_ << endl;

  t.commit ();
}
```

The only notable line in the above code is the creation of a session before the second transaction starts. As discussed in Chapter 11, "Session", a session acts as a cache of persistent objects. By creating a session before loading the `employee` objects we make sure that their `employer_` pointers point to the same `employer` object. Without a session, each `employee` would have ended up pointing to its own, private instance of the Example Inc employer.

As a general guideline, you should use a session when loading objects that have pointers to other persistent objects. A session makes sure that for a given object id, a single instance is shared among all other objects that relate to it.

We can also use data members from pointed-to objects in database queries (Chapter 4, "Querying the Database"). For each pointer in a persistent class, the query class defines a smart pointer-like member that contains members corresponding to the data members in the pointed-to object. We can then use the access via a pointer syntax (`->`) to refer to data members in pointed-to objects. For example, the query class for the `employee` object contains the `employer` member (its name is derived from the `employer_` pointer) which in turn contains the `name` member (its name is derived from the `employer::name_` data member of the pointed-to object). As a result, we can use the `query::employer->name` expression while querying the database for the `employee` objects. For example, the following transaction finds all the employees of Example Inc that have the Doe last name:

```
using query = odb::query<employee>;
using result = odb::result<employee>;

session s;
transaction t (db.begin ());

result r (db.query<employee> (
  query::employer->name == "Example Inc" && query::last == "Doe"));
```

```
for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cout << i->first_ << " " << i->last_ << endl;

t.commit ();
```

A query class member corresponding to a non-inverse (Section 6.2, "Bidirectional Relationships") object pointer can also be used as a normal member that has the id type of the pointed-to object. For example, the following query locates all the `employee` objects that don't have an associated `employer` object:

```
result r (db.query<employee> (query::employer.is_null ()));
```

An important concept to keep in mind when working with object relationships is the independence of persistent objects. In particular, when an object containing a pointer to another object is made persistent or is updated, the pointed-to object is not automatically persisted or updated. Rather, only a reference to the object (in the form of the object id) is stored for the pointed-to object in the database. The pointed-to object itself is a separate entity and should be made persistent or updated independently. By default, the same principle also applies to erasing pointed-to objects. That is, we have to make sure all the pointing objects are updated accordingly. However, in the case of erase, we can specify an alternative `on-delete` semantic as discussed in Section 14.4.15, "`on_delete`".

When persisting or updating an object containing a pointer to another object, the pointed-to object must have a valid object id. This, however, may not always be easy to achieve in complex relationships that involve objects with automatically assigned identifiers. In such cases it may be necessary to first persist an object with a pointer set to `NULL` and then, once the pointed-to object is made persistent and its identifier assigned, set the pointer to the correct value and update the object in the database.

Persistent object relationships can be divided into two groups: unidirectional and bidirectional. Each group in turn contains several configurations that vary depending on the cardinality of the sides of the relationship. All possible unidirectional and bidirectional configurations are discussed in the following sections.

6.1 Unidirectional Relationships

In unidirectional relationships we are only interested in navigating from object to object in one direction. Because there is no interest in navigating in the opposite direction, the cardinality of the other end of the relationship is unimportant. As a result, there are only two possible unidirectional relationships: to-one and to-many. Each of these relationships is described in the following sections. For sample code that shows how to work with these relationships, refer to the `relationship` example in the `odb-examples` package.

6.1.1 To-One Relationships

An example of a unidirectional to-one relationship is the employee-employer relationship (an employee has one employer). The following persistent C++ classes model this relationship:

```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<employer> employer_;
};
```

The corresponding database tables look like this:

```
CREATE TABLE employer (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employer VARCHAR (128) NOT NULL REFERENCES employer (name));
```

6.1.2 To-Many Relationships

An example of a unidirectional to-many relationship is the employee-project relationship (an employee can be involved in multiple projects). The following persistent C++ classes model this relationship:

```
#pragma db object
class project
{
  ...

  #pragma db id
  std::string name_;
};
```

```
#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db value_not_null unordered
  std::vector<shared_ptr<project>> projects_;
};
```

The corresponding database tables look like this:

```
CREATE TABLE project (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee_projects (
  object_id BIGINT UNSIGNED NOT NULL,
  value VARCHAR (128) NOT NULL REFERENCES project (name));
```

To obtain a more canonical database schema, the names of tables and columns above can be customized using ODB pragmas (Chapter 14, "ODB Pragma Language"). For example:

```
#pragma db object
class employee
{
  ...

  #pragma db value_not_null unordered \
              id_column("employee_id") value_column("project_name")
  std::vector<shared_ptr<project>> projects_;
};
```

The resulting `employee_projects` table would then look like this:

```
CREATE TABLE employee_projects (
  employee_id BIGINT UNSIGNED NOT NULL,
  project_name VARCHAR (128) NOT NULL REFERENCES project (name));
```

6.2 Bidirectional Relationships

In bidirectional relationships we are interested in navigating from object to object in both directions. As a result, each object class in a relationship contains a pointer to the other object. If smart pointers are used, then a weak pointer should be used as one of the pointers to avoid ownership cycles. For example:

```
class employee;

#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long long id_;

  weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<position> position_;
};
```

Note that when we establish a bidirectional relationship, we have to set both pointers consistently. One way to make sure that a relationship is always in a consistent state is to provide a single function that updates both pointers at the same time. For example:

```
#pragma db object
class position: public enable_shared_from_this<position>
{
  ...

  void
  fill (shared_ptr<employee> e)
  {
    employee_ = e;
    e->positions_ = shared_from_this ();
  }

private:
```

```
  weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

private:
  friend class position;

  #pragma db not_null
  shared_ptr<position> position_;
};
```

At the beginning of this chapter we examined how to use a session to make sure a single object is shared among all other objects pointing to it. With bidirectional relationships involving weak pointers the use of a session becomes even more crucial. Consider the following transaction that tries to load the `position` object from the above example without using a session:

```
transaction t (db.begin ())
shared_ptr<position> p (db.load<position> (1));
...
t.commit ();
```

When we load the `position` object, the `employee` object, which it points to, is also loaded. While `employee` is initially stored as `shared_ptr`, it is then assigned to the `employee_` member which is `weak_ptr`. Once the assignment is complete, the shared pointer goes out of scope and the only pointer that points to the newly loaded `employee` object is the `employee_` weak pointer. And that means the `employee` object is deleted immediately after being loaded. To help avoid such pathological situations ODB detects cases where a newly loaded object will immediately be deleted and throws the `odb::session_required` exception.

As the exception name suggests, the easiest way to resolve this problem is to use a session:

```
session s;
transaction t (db.begin ())
shared_ptr<position> p (db.load<position> (1));
...
t.commit ();
```

In our example, the session will maintain a shared pointer to the loaded `employee` object preventing its immediate deletion. Another way to resolve this problem is to avoid immediate loading of the pointed-to objects using lazy weak pointers. Lazy pointers are discussed in Section 6.4, "Lazy Pointers" later in this chapter.

Above, to model a bidirectional relationship in persistent classes, we used two pointers, one in each object. While this is a natural representation in C++, it does not translate to a canonical relational model. Consider the database schema generated for the above two classes:

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employee BIGINT UNSIGNED REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

While this database schema is valid, it is unconventional. We have a reference from a row in the `position` table to a row in the `employee` table. We also have a reference from this same row in the `employee` table back to the row in the `position` table. From the relational point of view, one of these references is redundant since in SQL we can easily navigate in both directions using just one of these references.

To eliminate redundant database schema references we can use the `inverse` pragma (Section 14.4.14, "`inverse`") which tells the ODB compiler that a pointer is the inverse side of a bidirectional relationship. Either side of a relationship can be made inverse. For example:

```
#pragma db object
class position
{
  ...

  #pragma db inverse(position_)
  weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

  #pragma db not_null
  shared_ptr<position> position_;
};
```

The resulting database schema looks like this:

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

As you can see, an inverse member does not have a corresponding column (or table, in case of an inverse container of pointers) and, from the point of view of database operations, is effectively read-only. The only way to change a bidirectional relationship with an inverse side is to set its direct (non-inverse) pointer. Also note that an ordered container (Section 5.1, "Ordered Containers") of pointers that is an inverse side of a bidirectional relationship is always treated as unordered (Section 14.4.19, "`unordered`") because the contents of such a container are implicitly built from the direct side of the relationship which does not contain the element order (index).

There are three distinct bidirectional relationships that we will cover in the following sections: one-to-one, one-to-many, and many-to-many. We will only talk about bidirectional relationships with inverse sides since they result in canonical database schemas. For sample code that shows how to work with these relationships, refer to the `inverse` example in the `odb-examples` package.

6.2.1 One-to-One Relationships

An example of a bidirectional one-to-one relationship is the presented above employee-position relationship (an employee fills one position and a position is filled by one employee). The following persistent C++ classes model this relationship:

```
class employee;

#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db inverse(position_)
  weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<position> position_;
};
```

The corresponding database tables look like this:

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

If instead the other side of this relationship is made inverse, then the database tables will change as follows:

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employee BIGINT UNSIGNED REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);
```

6.2.2 One-to-Many Relationships

An example of a bidirectional one-to-many relationship is the employer-employee relationship (an employer has multiple employees and an employee is employed by one employer). The following persistent C++ classes model this relationship:

```
class employee;

#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name_;

  #pragma db value_not_null inverse(employer_)
  std::vector<weak_ptr<employee>> employees_
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<employer> employer_;
};
```

The corresponding database tables differ significantly depending on which side of the relationship is made inverse. If the *one* side (`employer`) is inverse as in the code above, then the resulting database schema looks like this:

```
CREATE TABLE employer (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employer VARCHAR (128) NOT NULL REFERENCES employer (name));
```

If instead the *many* side (`employee`) of this relationship is made inverse, then the database tables will change as follows:

```
CREATE TABLE employer (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE employer_employees (
  object_id VARCHAR (128) NOT NULL REFERENCES employer (name),
  value BIGINT UNSIGNED NOT NULL REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);
```

6.2.3 Many-to-Many Relationships

An example of a bidirectional many-to-many relationship is the employee-project relationship (an employee can work on multiple projects and a project can have multiple participating employees). The following persistent C++ classes model this relationship:

```
class employee;

#pragma db object
class project
{
  ...

  #pragma db id
  std::string name_;

  #pragma db value_not_null inverse(projects_)
  std::vector<weak_ptr<employee>> employees_;
};

#pragma db object
class employee
{
  ...
```

```
  #pragma db id
  unsigned long long id_;

  #pragma db value_not_null unordered
  std::vector<shared_ptr<project>> projects_;
};
```

The corresponding database tables look like this:

```
CREATE TABLE project (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee_projects (
  object_id BIGINT UNSIGNED NOT NULL REFERENCES employee (id),
  value VARCHAR (128) NOT NULL REFERENCES project (name));
```

If instead the other side of this relationship is made inverse, then the database tables will change as follows:

```
CREATE TABLE project (
  name VARCHAR (128) NOT NULL PRIMARY KEY);

CREATE TABLE project_employees (
  object_id VARCHAR (128) NOT NULL REFERENCES project (name),
  value BIGINT UNSIGNED NOT NULL REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);
```

6.3 Circular Relationships

A relationship between two persistent classes is circular if each of them references the other. Bidirectional relationships are always circular. A unidirectional relationship combined with inheritance (Chapter 8, "Inheritance") can also be circular. For example, the `employee` class could derive from `person` which, in turn, could contain a pointer to `employee`.

We don’t need to do anything extra if persistent classes with circular dependencies are defined in the same header file. Specifically, ODB will make sure that the database tables and foreign key constraints are created in the correct order. As a result, unless you have good reasons not to, it is recommended that you keep persistent classes with circular dependencies in the same header file.

If you have to keep such classes in separate header files, then there are two extra steps that you may need to take in order to use these classes with ODB. Consider again the example from Section 6.2.1, "One-to-One Relationships" but this time with the classes defined in separate

headers:

```
// position.hxx
//
class employee;

#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db inverse(position_)
  weak_ptr<employee> employee_;
};

// employee.hxx
//
#include "position.hxx"

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<position> position_;
};
```

Note that the `position.hxx` header contains only the forward declaration for `employee`. While this is sufficient to define a valid, from the C++ point of view, `position` class, the ODB compiler needs to "see" the definitions of the pointed-to persistent classes. There are several ways we can fulfil this requirement. The easiest is to simply include `employee.hxx` at the end of `position.hxx`:

```
// position.hxx
//
class employee;

#pragma db object
class position
{
```

```
  ...
};

#include "employee.hxx"
```

We can also limit this inclusion only to the time when `position.hxx` is compiled with the ODB compiler:

```
// position.hxx
//

...

#ifdef ODB_COMPILER
#  include "employee.hxx"
#endif
```

Finally, if we don't want to modify `position.hxx`, then we can add `employee.hxx` to the ODB compilation process with the `--odb-epilogue` option. For example:

```
odb ... --odb-epilogue "#include \"employee.hxx\"" position.hxx
```

Note also that in this example we didn't have to do anything extra for `employee.hxx` because it already includes `position.hxx`. However, if instead it relied only on the forward declaration of the `position` class, then we would have to handle it in the same way as `position.hxx`.

The other difficulty with separately defined classes involving circular relationships has to do with the correct order of foreign key constraint creation in the generated database schema. In the above example, if we generate the database schema as standalone SQL files, then we will end up with two such files: `position.sql` and `employee.sql`. If we try to execute `employee.sql` first, then we will get an error indicating that the table corresponding to the `position` class and referenced by the foreign key constraint corresponding to the `position_` pointer does not yet exist.

Note that there is no such problem if the database schema is embedded in the generated C++ code instead of being produced as standalone SQL files. In this case, the ODB compiler is able to ensure the correct creation order even if the classes are defined in separate header files.

In certain cases, for example, a bidirectional relationship with an inverse side, this problem can be resolved by executing the database schema creation files in the correct order. In our example, this would be `position.sql` first and `employee.sql` second. However, this approach doesn't scale beyond simple object models.

A more robust solution to this problem is to generate the database schema for all the persistent classes into a single SQL file. This way, the ODB compiler can again ensure the correct creation order of tables and foreign keys. To instruct the ODB compiler to produce a combined schema file for several headers we can use the `--generate-schema-only` and `--at-once` options. For example:

```
odb ... --generate-schema-only --at-once --input-name company \
position.hxx employee.hxx
```

The result of the above command is a single `company.sql` file (the name is derived from the `--input-name` value) that contains the database creation code for both `position` and `employee` classes.

6.4 Lazy Pointers

Consider again the bidirectional, one-to-many employer-employee relationship that was presented earlier in this chapter:

```
class employee;

#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name_;

  #pragma db value_not_null inverse(employer_)
  std::vector<weak_ptr<employee>> employees_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long long id_;

  #pragma db not_null
  shared_ptr<employer> employer_;
};
```

Consider also the following transaction which obtains the employer name given the employee id:

```
unsigned long long id = ...
string name;

session s;
transaction t (db.begin ());

shared_ptr<employee> e (db.load<employee> (id));
name = e->employer_->name_;

t.commit ();
```

While this transaction looks very simple, it actually does a lot more than what meets the eye and is necessary. Consider what happens when we load the `employee` object: the `employer_` pointer is also automatically loaded which means the `employer` object corresponding to this employee is also loaded. But the `employer` object in turn contains the list of pointers to all the employees, which are also loaded. As a result, when object relationships are involved, a simple transaction like the above can load many more objects than is necessary.

To overcome this problem ODB offers finer grained control over the relationship loading in the form of lazy pointers. A lazy pointer does not automatically load the pointed-to object when the containing object is loaded. Instead, we have to explicitly load the pointed-to object if and when we need to access it.

The ODB runtime library provides lazy counterparts for all the supported pointers, namely: `odb::lazy_shared_ptr`/`lazy_weak_ptr` for C++11 `std::shared_ptr`/`weak_ptr`, `odb::lazy_unique_ptr` for C++11 `std::unique_ptr`, `odb::lazy_auto_ptr` for C++98/03 `std::auto_ptr`, and `odb::lazy_ptr` for raw pointers. All these lazy pointers are defined in the `<odb/lazy-ptr.hxx>` header. The ODB profile libraries also provide lazy pointer implementations for smart pointers from popular frameworks and libraries (Part III, "Profiles").

While we will discuss the interface of lazy pointers in more detail shortly, the most commonly used extra function provided by these pointers is `load()`. This function loads the pointed-to object if it hasn't already been loaded. After the call to this function, the lazy pointer can be used in the the same way as its eager counterpart. The `load()` function also returns the eager pointer, in case you need to pass it around. For a lazy weak pointer, the `load()` function also locks the pointer.

The following example shows how we can change our employer-employee relationship to use lazy pointers. Here we choose to use lazy pointers for both sides of the relationship.

```
class employee;

#pragma db object
class employer
{
```

```
  ...

  #pragma db value_not_null inverse(employer_)
  std::vector<lazy_weak_ptr<employee>> employees_;
};

#pragma db object
class employee
{
  ...

  #pragma db not_null
  lazy_shared_ptr<employer> employer_;
};
```

And the transaction is changed like this:

```
unsigned long long id = ...
string name;

session s;
transaction t (db.begin ());

shared_ptr<employee> e (db.load<employee> (id));
e->employer_.load ();
name = e->employer_->name_;

t.commit ();
```

As a general guideline we recommend that you make at least one side of a bidirectional relationship lazy, especially for relationships with a *many* side.

A lazy pointer implementation mimics the interface of its eager counterpart which can be used once the pointer is loaded. It also adds a number of additional functions that are specific to the lazy loading functionality. Overall, the interface of a lazy pointer follows this general outline:

```
template <class T>
class lazy_ptr
{
public:
  //
  // The eager pointer interface.
  //

  // Initialization/assignment from an eager pointer to a
  // transient object.
  //
public:
  template <class Y> lazy_ptr (const eager_ptr<Y>&);
  template <class Y> lazy_ptr& operator= (const eager_ptr<Y>&);
```

```
  // Lazy loading interface.
  //
public:
  //  NULL      loaded()
  //
  //  true       true      NULL pointer to transient object
  //  false      true      valid pointer to persistent object
  //  true       false     unloaded pointer to persistent object
  //  false      false     valid pointer to transient object
  //
  bool loaded () const;

  eager_ptr<T> load () const;

  // Unload the pointer. For transient objects this function is
  // equivalent to reset().
  //
  void unload () const;

  // Get the underlying eager pointer. If this is an unloaded pointer
  // to a persistent object, then the returned pointer will be NULL.
  //
  eager_ptr<T> get_eager () const;

  // Initialization with a persistent loaded object.
  //
  template <class Y> lazy_ptr (database&, Y*);
  template <class Y> lazy_ptr (database&, const eager_ptr<Y>&);

  template <class Y> void reset (database&, Y*);
  template <class Y> void reset (database&, const eager_ptr<Y>&);

  // Initialization with a persistent unloaded object.
  //
  template <class ID> lazy_ptr (database&, const ID&);

  template <class ID> void reset (database&, const ID&);

  // Query object id and database of a persistent object.
  //
  template <class O /* = T */>
  // C++11: template <class O = T>
  object_traits<O>::id_type object_id () const;

  odb::database& database () const;
};
```

Note that to initialize a lazy pointer to a persistent object from its eager pointer one must use the constructor or `reset()` function with the database as its first argument. The database is required to support comparison of unloaded lazy pointers to persistent objects.

In a lazy weak pointer interface, the `load()` function returns the *strong* (shared) eager pointer. The following transaction demonstrates the use of a lazy weak pointer based on the `employer` and `employee` classes presented earlier.

```
using employees = std::vector<lazy_weak_ptr<employee>>;

session s;
transaction t (db.begin ());

shared_ptr<employer> er (db.load<employer> ("Example Inc"));
employees& es (er->employees ());

for (employees::iterator i (es.begin ()); i != es.end (); ++i)
{
  // We are only interested in employees with object id less than
  // 100.
  //
  lazy_weak_ptr<employee>& lwp (*i);

  if (lwp.object_id<employee> () < 100)
  // C++11: if (lwp.object_id () < 100)
  {
    shared_ptr<employee> e (lwp.load ()); // Load and lock.
    cout << e->first_ << " " << e->last_ << endl;
  }
}

t.commit ();
```

Notice that inside the for-loop we use a reference to the lazy weak pointer instead of making a copy. This is not merely to avoid a copy. When a lazy pointer is loaded, all other lazy pointers that point to the same object do not automatically become loaded (though an attempt to load such copies will result in them pointing to the same object, provided the same session is still in effect). By using a reference in the above transaction we make sure that we load the pointer that is contained in the `employer` object. This way, if we later need to re-examine this `employee` object, the pointer will already be loaded.

As another example, suppose we want to add an employee to Example Inc. The straightforward implementation of this transaction is presented below:

```
session s;
transaction t (db.begin ());

shared_ptr<employer> er (db.load<employer> ("Example Inc"));
```

```
shared_ptr<employee> e (new employee ("John", "Doe"));

e->employer_ = er;
er->employees ().push_back (e);

db.persist (e);
t.commit ();
```

Notice here that we didn't have to update the employer object in the database since the `employees_` list of pointers is an inverse side of a bidirectional relationship and is effectively read-only, from the persistence point of view.

A faster implementation of this transaction, that avoids loading the employer object, relies on the ability to initialize an *unloaded* lazy pointer with the database where the object is stored as well as its identifier:

```
lazy_shared_ptr<employer> er (db, std::string ("Example Inc"));
shared_ptr<employee> e (new employee ("John", "Doe"));

e->employer_ = er;

session s;
transaction t (db.begin ());

db.persist (e);

t.commit ();
```

For the interaction of lazy pointers with lazy-loaded object sections, refer to Section 9.3, "Sections and Lazy Pointers".

6.5 Using Custom Smart Pointers

While the ODB runtime and profile libraries provide support for the majority of widely-used pointers, it is also easy to add support for a custom smart pointer.

To achieve this you will need to implement the `pointer_traits` class template specialization for your pointer. The first step is to determine the pointer kind since the interface of the `pointer_traits` specialization varies depending on the pointer kind. The supported pointer kinds are: *raw* (raw pointer or equivalent, that is, unmanaged), *unique* (smart pointer that doesn't support sharing), *shared* (smart pointer that supports sharing), and *weak* (weak counterpart of the shared pointer). Any of these pointers can be lazy, which also affects the interface of the `pointer_traits` specialization.

Once you have determined the pointer kind for your smart pointer, use a specialization for one of the standard pointers found in the common ODB runtime library (`libodb`) as a base for your own implementation.

Once the pointer traits specialization is ready, you will need to include it into the ODB compilation process using the `--odb-epilogue` option and into the generated header files with the `--hxx-prologue` option. As an example, suppose we have the `smart_ptr` smart pointer for which we have the traits specialization implemented in the `smart-ptr-traits.hxx` file. Then, we can create an ODB compiler options file for this pointer and save it to `smart-ptr.options`:

```
# Options file for smart_ptr.
#
--odb-epilogue '#include "smart-ptr-traits.hxx"'
--hxx-prologue '#include "smart-ptr-traits.hxx"'
```

Now, whenever we compile a header file that uses `smart_ptr`, we can specify the following command line option to make sure it is recognized by the ODB compiler as a smart pointer and the traits file is included in the generated code:

```
--options-file smart-ptr.options
```

It is also possible to implement a lazy counterpart for your smart pointer. The ODB runtime library provides a class template that encapsulates the object id management and loading functionality that is needed to implement a lazy pointer. All you need to do is wrap it with an interface that mimics your smart pointer. Using one of the existing lazy pointer implementations (either from the ODB runtime library or one of the profile libraries) as a base for your implementation is the easiest way to get started.

7 Value Types

In Section 3.1, "Concepts and Terminology" we have already discussed the notion of values and value types as well as the distinction between simple and composite values. This chapter covers simple and composite value types in more detail.

7.1 Simple Value Types

A simple value type is a fundamental C++ type or a class type that is mapped to a single database column. For each supported database system the ODB compiler provides a default mapping to suitable database types for most fundamental C++ types, such as `int` or `float` as well as some class types, such as `std::string`. For more information about the default mapping for each database system refer to Part II, Database Systems. We can also provide a custom mapping for these or our own value types using the `db type` pragma (Section 14.3.1, "`type`").

7.2 Composite Value Types

A composite value type is a `class` or `struct` type that is mapped to more than one database column. To declare a composite value type we use the `db value` pragma, for example:

```
#pragma db value
class basic_name
{
  ...

  std::string first_;
  std::string last_;
};
```

The complete version of the above code fragment and the other code samples presented in this section can be found in the `composite` example in the `odb-examples` package.

A composite value type does not have to define a default constructor, unless it is used as an element of a container. In this case the default constructor can be made private provided we also make the `odb::access` class, defined in the `<odb/core.hxx>` header, a friend of this value type. For example:

```
#include <odb/core.hxx>

#pragma db value
class basic_name
{
public:
  basic_name (const std::string& first, const std::string& last);
```

```
  ...

private:
  friend class odb::access;

  basic_name () {} // Needed for storing basic_name in containers.

  ...
};
```

The ODB compiler also needs access to the non-transient (Section 14.4.11, "`transient`") data members of a composite value type. It uses the same mechanisms as for persistent classes which are discussed in Section 3.2, "Declaring Persistent Objects and Values".

The members of a composite value can be other value types (either simple or composite), containers (Chapter 5, "Containers"), and pointers to objects (Chapter 6, "Relationships"). Similarly, a composite value type can be used in object members, as an element of a container, and as a base for another composite value type. In particular, composite value types can be used as element types in set containers (Section 5.2, "Set and Multiset Containers") and as key types in map containers (Section 5.3, "Map and Multimap Containers"). A composite value type that is used as an element of a container cannot contain other containers since containers of containers are not allowed. The following example illustrates some of the possible use cases:

```
#pragma db value
class basic_name
{
  ...

  std::string first_;
  std::string last_;
};

using basic_names = std::vector<basic_name>;

#pragma db value
class name_extras
{
  ...

  std::string nickname_;
  basic_names aliases_;
};

#pragma db value
class name: public basic_name
{
  ...

  std::string title_;
```

```
  name_extras extras_;
};

#pragma db object
class person
{
  ...

  name name_;
};
```

A composite value type can be defined inside a persistent class, view, or another composite value and even made private, provided we make `odb::access` a friend of the containing class, for example:

```
#pragma db object
class person
{
  ...

  #pragma db value
  class name
  {
    ...

    std::string first_;
    std::string last_;
  };

  name name_;
};
```

A composite value type can also be defined as an instantiation of a C++ class template, for example:

```
template <typename T>
struct point
{
  T x;
  T y;
  T z;
};

using int_point = point<int>;
#pragma db value(int_point)

#pragma db object
class object
{
```

```
  ...

  int_point center_;
};
```

Note that the database support code for such a composite value type is generated when compiling the header containing the `db value` pragma and not the header containing the template definition or the `using` alias. This allows us to use templates defined in other files, such as `std::pair` defined in the `utility` standard header file:

```
#include <utility> // std::pair

using phone_numbers = std::pair<std::string, std::string>;
#pragma db value(phone_numbers)

#pragma db object
class person
{
  ...

  phone_numbers phone_;
};
```

We can also use data members from composite value types in database queries (Chapter 4, "Querying the Database"). For each composite value in a persistent class, the query class defines a nested member that contains members corresponding to the data members in the value type. We can then use the member access syntax (.) to refer to data members in value types. For example, the query class for the `person` object presented above contains the `name` member (its name is derived from the `name_` data member) which in turn contains the `extras` member (its name is derived from the `name::extras_` data member of the composite value type). This process continues recursively for nested composite value types and, as a result, we can use the `query::name.extras.nickname` expression while querying the database for the `person` objects. For example:

```
using query = odb::query<person>;
using result = odb::result<person>;

transaction t (db.begin ());

result r (db.query<person> (
  query::name.extras.nickname == "Squeaky"));

...

t.commit ();
```

7.2.1 Composite Object Ids

An object id can be of a composite value type, for example:

```
#pragma db value
class name
{
  ...

  std::string first_;
  std::string last_;
};

#pragma db object
class person
{
  ...

  #pragma db id
  name name_;
};
```

However, a value type that can be used as an object id has a number of restrictions. Such a value type cannot have container, object pointer, or read-only data members. It also must be default-constructible, copy-constructible, and copy-assignable. Furthermore, if the persistent class in which this composite value type is used as object id has session support enabled (Chapter 11, "Session"), then it must also implement the less-than comparison operator (`operator<`).

7.2.2 Composite Value Column and Table Names

Customizing a column name for a data member of a simple value type is straightforward: we simply specify the desired name with the `db column` pragma (Section 14.4.9, "`column`"). For composite value types things are slightly more complex since they are mapped to multiple columns. Consider the following example:

```
#pragma db value
class name
{
  ...

  std::string first_;
  std::string last_;
};

#pragma db object
class person
{
  ...
```

```
  #pragma db id auto
  unsigned long long id_;

  name name_;
};
```

The column names for the `first_` and `last_` members are constructed by using the sanitized name of the `person::name_` member as a prefix and the names of the members in the value type (`first_` and `last_`) as suffixes. As a result, the database schema for the above classes will look like this:

```
CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  name_first TEXT NOT NULL,
  name_last TEXT NOT NULL);
```

We can customize both the prefix and the suffix using the `db column` pragma as shown in the following example:

```
#pragma db value
class name
{
  ...

  #pragma db column("first_name")
  std::string first_;

  #pragma db column("last_name")
  std::string last_;
};

#pragma db object
class person
{
  ...

  #pragma db column("person_")
  name name_;
};
```

The database schema changes as follows:

```
CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  person_first_name TEXT NOT NULL,
  person_last_name TEXT NOT NULL);
```

We can also make the column prefix empty, for example:

```
#pragma db object
class person
{
  ...

  #pragma db column("")
  name name_;
};
```

This will result in the following schema:

```
CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL);
```

The same principle applies when a composite value type is used as an element of a container, except that instead of `db column`, either the `db value_column` (Section 14.4.36, "`value_column`") or `db key_column` (Section 14.4.35, "`key_column`") pragmas are used to specify the column prefix.

When a composite value type contains a container, an extra table is used to store its elements (Chapter 5, "Containers"). The names of such tables are constructed in a way similar to the column names, except that by default both the object name and the member name are used as a prefix. For example:

```
#pragma db value
class name
{
  ...

  std::string first_;
  std::string last_;
  std::vector<std::string> nicknames_;
};

#pragma db object
class person
{
  ...

  name name_;
};
```

The corresponding database schema will look like this:

```
CREATE TABLE person_name_nicknames (
  object_id BIGINT UNSIGNED NOT NULL,
  index BIGINT UNSIGNED NOT NULL,
  value TEXT NOT NULL)

CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  name_first TEXT NOT NULL,
  name_last TEXT NOT NULL);
```

To customize the container table name we can use the `db table` pragma (Section 14.4.20, "`table`"), for example:

```
#pragma db value
class name
{
  ...

  #pragma db table("nickname")
  std::vector<std::string> nicknames_;
};

#pragma db object
class person
{
  ...

  #pragma db table("person_")
  name name_;
};
```

This will result in the following schema changes:

```
CREATE TABLE person_nickname (
  object_id BIGINT UNSIGNED NOT NULL,
  index BIGINT UNSIGNED NOT NULL,
  value TEXT NOT NULL)
```

Similar to columns, we can make the table prefix empty.

7.3 Pointers and `NULL` Value Semantics

Relational database systems have a notion of the special `NULL` value that is used to indicate the absence of a valid value in a column. While by default ODB maps values to columns that do not allow `NULL` values, it is possible to change that with the `db null` pragma (Section 14.4.6, "`null`/`not_null`").

To properly support the NULL semantics, the C++ value type must have a notion of a NULL value or a similar special state concept. Most basic C++ types, such as int or std::string, do not have this notion and therefore cannot be used directly for NULL-enabled data members (in the case of a NULL value being loaded from the database, such data members will be default-initialized).

To allow the easy conversion of value types that do not support the NULL semantics into the ones that do, ODB provides the odb::nullable class template. It allows us to wrap an existing C++ type into a container-like class that can either be NULL or contain a value of the wrapped type. ODB also automatically enables the NULL values for data members of the odb::nullable type. For example:

```
#include <odb/nullable.hxx>

#pragma db object
class person
{
  ...

  std::string first_;                     // TEXT NOT NULL
  odb::nullable<std::string> middle_;     // TEXT NULL
  std::string last_;                      // TEXT NOT NULL
};
```

The odb::nullable class template is defined in the <odb/nullable.hxx> header file and has the following interface:

```
namespace odb
{
  template <typename T>
  class nullable
  {
  public:
    using value_type = T;

    nullable ();
    nullable (const T&);
    nullable (const nullable&);
    template <typename Y> explicit nullable (const nullable<Y>&);

    nullable& operator= (const T&);
    nullable& operator= (const nullable&);
    template <typename Y> nullable& operator= (const nullable<Y>&);

    void swap (nullable&);

    // Accessor interface.
    //
    bool null () const;
```

```
    T&       get ();
    const T& get () const;

    // Pointer interface.
    //
    operator bool_convertible () const;

    T*       operator-> ();
    const T* operator-> () const;

    T&       operator* ();
    const T& operator* () const;

    // Reset to the NULL state.
    //
    void reset ();
  };
}
```

The following example shows how we can use this interface:

```
  nullable<string> ns;

  // Using the accessor interface.
  //
  if (ns.null ())
  {
    s = "abc";
  }
  else
  {
    string s (ns.get ());
    ns.reset ();
  }

  // The same using the pointer interface.
  //
  if (!ns)
  {
    s = "abc";
  }
  else
  {
    string s (*ns);
    ns.reset ();
  }
```

The `odb::nullable` class template requires the wrapped type to have public default and copy constructors as well as the copy assignment operator. Note also that the `odb::nullable` implementation is not the most efficient in that it always contains a fully constructed value of the wrapped type. This is normally not a concern for simple types such as the C++ fundamental types or `std::string`. However, it may become an issue for more complex types. In such cases you may want to consider using a more efficient implementation of the *optional value* concept such as the `optional` class template from Boost (Section 23.4, "Optional Library").

Another common C++ representation of a value that can be `NULL` is a pointer. ODB will automatically handle data members that are pointers to values, however, it will not automatically enable `NULL` values for such data members, as is the case for `odb::nullable`. Instead, if the `NULL` value is desired, we will need to enable it explicitly using the `db null` pragma. For example:

```
#pragma db object
class person
{
  ...

  std::string first_;

  #pragma db null
  std::unique_ptr<std::string> middle_;

  std::string last_;
};
```

The ODB compiler includes built-in support for using `std::auto_ptr` (C++98/03 only), `std::unique_ptr` (C++11), and `std::shared_ptr` (C++11) as pointers to values. Plus, ODB profile libraries, that are available for commonly used frameworks and libraries (such as Boost and Qt), provide support for smart pointers found in these frameworks and libraries (Part III, "Profiles").

ODB also supports the `NULL` semantics for composite values. In the relational database the `NULL` composite value is translated to `NULL` values for all the simple data members of this composite value. For example:

```
#pragma db value
struct name
{
  std::string first_;
  odb::nullable<std::string> middle_;
  std::string last_;
};

#pragma db object
class person
```

```
{
  ...
  odb::nullable<name> name_;
};
```

ODB does not support the `NULL` semantics for containers. This also means that a composite value that contains a container cannot be `NULL`. With this limitation in mind, we can still use smart pointers in data members of container types. The only restriction is that these pointers must not be `NULL`. For example:

```
#pragma db object
class person
{
  ...

  std::unique_ptr<std::vector<std::string>> aliases_;
};
```

8 Inheritance

In C++ inheritance can be used to achieve two different goals. We can employ inheritance to reuse common data and functionality in multiple classes. For example:

```
class person
{
public:
  const std::string& first () const;
  const std::string& last () const;

private:
  std::string first_;
  std::string last_;
};

class employee: public person
{
  ...
};

class contractor: public person
{
  ...
};
```

In the above example both the `employee` and `contractor` classes inherit the `first_` and `last_` data members as well as the `first()` and `last()` accessors from the `person` base class.

A common trait of this inheritance style, referred to as *reuse inheritance* from now on, is the lack of virtual functions and a virtual destructor in the base class. Also with this style the application code is normally written in terms of the derived classes instead of the base.

The second way to utilize inheritance in C++ is to provide polymorphic behavior through a common interface. In this case the base class defines a number of virtual functions and, normally, a virtual destructor while the derived classes provide specific implementations of these virtual functions. For example:

```
class person
{
public:
  enum employment_status
  {
    unemployed,
    temporary,
    permanent,
    self_employed
```

```
  };

  virtual employment_status
  employment () const = 0;

  virtual
  ~person ();
};

class employee: public person
{
public:
  virtual employment_status
  employment () const
  {
    return temporary_ ? temporary : permanent;
  }

private:
  bool temporary_;
};

class contractor: public person
{
public:
  virtual employment_status
  employment () const
  {
    return self_employed;
  }
};
```

With this inheritance style, which we will call *polymorphism inheritance*, the application code normally works with derived classes via the base class interface. Note also that it is very common to mix both styles in the same hierarchy. For example, the above two code fragments can be combined so that the `person` base class provides the common data members and functions as well as defines the polymorphic interface.

The following sections describe the available strategies for mapping reuse and polymorphism inheritance styles to a relational data model. Note also that the distinction between the two styles is conceptual rather than formal. For example, it is possible to treat a class hierarchy that defines virtual functions as a case of reuse inheritance if this results in the desired database mapping and semantics.

Generally, classes that employ reuse inheritance are mapped to completely independent entities in the database. They use different object id spaces and should always be passed to and returned from the database operations as pointers or references to derived types. In other words, from the persistence point of view, such classes behave as if the data members from the base classes were

copied verbatim into the derived ones.

In contrast, classes that employ polymorphism inheritance share the object id space and can be passed to and returned from the database operations *polymorphically* as pointers or references to the base class.

For both inheritance styles it is sometimes desirable to prevent instances of a base class from being stored in the database. To achieve this a persistent class can be declared abstract using the `db abstract` pragma (Section 14.1.3, "`abstract`"). Note that a *C++-abstract* class, or a class that has one or more pure virtual functions and therefore cannot be instantiated, is also *database-abstract*. However, a database-abstract class is not necessarily C++-abstract. The ODB compiler automatically treats C++-abstract classes as database-abstract.

8.1 Reuse Inheritance

Each non-abstract class from the reuse inheritance hierarchy is mapped to a separate database table that contains all its data members, including those inherited from base classes. An abstract persistent class does not have to define an object id, nor a default constructor, and it does not have a corresponding database table. An abstract class cannot be a pointed-to object in a relationship. Multiple inheritance is supported as long as each base class is only inherited once. The following example shows a persistent class hierarchy employing reuse inheritance:

```
// Abstract person class. Note that it does not declare the
// object id.
//
#pragma db object abstract
class person
{
  ...

  std::string first_;
  std::string last_;
};

// Abstract employee class. It derives from the person class and
// declares the object id for all the concrete employee types.
//
#pragma db object abstract
class employee: public person
{
  ...

  #pragma db id auto
  unsigned long long id_;
};

// Concrete permanent_employee class. Note that it doesn't define
```

```
// any data members of its own.
//
#pragma db object
class permanent_employee: public employee
{
  ...
};

// Concrete temporary_employee class. It adds the employment
// duration in months.
//
#pragma db object
class temporary_employee: public employee
{
  ...

  unsigned long duration_;
};

// Concrete contractor class. It derives from the person class
// (and not employee; an independent contractor is not considered
// an employee). We use the contractor's external email address
// as the object id.
//
#pragma db object
class contractor: public person
{
  ...

  #pragma db id
  std::string email_;
};
```

The sample database schema for this hierarchy is shown below.

```
CREATE TABLE permanent_employee (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT);

CREATE TABLE temporary_employee (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
  duration BIGINT UNSIGNED NOT NULL);

CREATE TABLE contractor (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  email VARCHAR (128) NOT NULL PRIMARY KEY);
```

The complete version of the code presented in this section is available in the `inheritance/reuse` example in the `odb-examples` package.

8.2 Polymorphism Inheritance

There are three general approaches to mapping a polymorphic class hierarchy to a relational database. These are *table-per-hierarchy*, *table-per-difference*, and *table-per-class*. With the table-per-hierarchy mapping, all the classes in a hierarchy are stored in a single, "wide" table. `NULL` values are stored in columns corresponding to data members of derived classes that are not present in any particular instance.

In the table-per-difference mapping, each class is mapped to a separate table. For a derived class, this table contains only columns corresponding to the data members added by this derived class.

Finally, in the table-per-class mapping, each class is mapped to a separate table. For a derived class, this table contains columns corresponding to all the data members, from this derived class all the way down to the root of the hierarchy.

The table-per-difference mapping is generally considered as having the best balance of flexibility, performance, and space efficiency. It also results in a more canonical relational database model compared to the other two approaches. As a result, this is the mapping currently implemented in ODB. Other mappings may be supported in the future. Note that multiple polymorphism inheritance or mixing polymorphism and reuse inheritance is not supported.

A pointer or reference to an ordinary, non-polymorphic object has just one type — the class type of that object. When we start working with polymorphic objects, there are two types to consider: the *static type*, or the declaration type of a reference or pointer, and the object's actual or *dynamic type*. An example will help illustrate the difference:

```
class person {...};
class employee: public person {...};

person p;
employee e;

person& r1 (p);
person& r2 (e);

unique_ptr<person> p1 (new employee);
```

In the above example, the `r1` reference's both static and dynamic types are `person`. In contrast, the `r2` reference's static type is `person` while its dynamic type (the actual object that it refers to) is `employee`. Similarly, `p1` points to the object of the `person` static type but `employee` dynamic type.

In C++, the primary mechanisms for working with polymorphic objects are virtual functions. We call a virtual function only knowing the object's static type, but the version corresponding to the object's dynamic type is automatically executed. This is the essence of runtime polymorphism support in C++: we can operate in terms of a base class interface but get the derived class' behavior. Similarly, the essence of the runtime polymorphism support in ODB is to allow us to persist, load, update, and query in terms of the base class interface but have the derived class actually stored in the database.

To declare a persistent class as polymorphic we use the `db polymorphic` pragma. We only need to declare the root class of a hierarchy as polymorphic; ODB will treat all the derived classes as polymorphic automatically. For example:

```
#pragma db object polymorphic
class person
{
  ...

  virtual
  ~person () = 0; // Automatically abstract.

  #pragma db id auto
  unsigned long long id_;

  std::string first_;
  std::string last_;
};

#pragma db object
class employee: public person
{
  ...

  bool temporary_;
};

#pragma db object
class contractor: public person
{

  std::string email_;
};
```

A persistent class hierarchy declared polymorphic must also be polymorphic in the C++ sense, that is, the root class must declare or inherit at least one virtual function. It is recommended that the root class also declares a virtual destructor. The root class of the polymorphic hierarchy must contain the data member designated as object id (a persistent class without an object id cannot be polymorphic). Note also that, unlike reuse inheritance, abstract polymorphic classes have a table in the database, just like non-abstract classes.

Persistent classes in the same polymorphic hierarchy must use the same kind of object pointer (Section 3.3, "Object and View Pointers"). If the object pointer for the root class is specified as a template or using the special raw pointer syntax (`*`), then the ODB compiler will automatically use the same object pointer for all the derived classes. For example:

```
#pragma db object polymorphic pointer(std::shared_ptr)
class person
{
  ...
};

#pragma db object // Object pointer is std::shared_ptr<employee>.
class employee: public person
{
  ...
};

#pragma db object // Object pointer is std::shared_ptr<contractor>.
class contractor: public person
{
  ...
};
```

Similarly, if we enable or disable session support (Chapter 11, "Session") for the root class, then the ODB compiler will automatically enable or disable it for all the derived classes.

For polymorphic persistent classes, all the database operations can be performed on objects with different static and dynamic types. Similarly, operations that load persistent objects from the database (`load()`, `query()`, etc.), can return objects with different static and dynamic types. For example:

```
unsigned long long id1, id2;

// Persist.
//
{
  shared_ptr<person> p1 (new employee (...));
  shared_ptr<person> p2 (new contractor (...));

  transaction t (db.begin ());
  id1 = db.persist (p1); // Stores employee.
  id2 = db.persist (p2); // Stores contractor.
  t.commit ();
}

// Load.
//
{
  shared_ptr<person> p;
```

```
  transaction t (db.begin ());
  p = db.load<person> (id1); // Loads employee.
  p = db.load<person> (id2); // Loads contractor.
  t.commit ();
}

// Query.
//
{
  using query = odb::query<person>;
  using result = odb::result<person>;

  transaction t (db.begin ());

  result r (db.query<person> (query::last == "Doe"));

  for (result::iterator i (r.begin ()); i != r.end (); ++i)
  {
    person& p (*i); // Can be employee or contractor.
  }

  t.commit ();
}

// Update.
//
{
  shared_ptr<person> p;
  shared_ptr<employee> e;

  transaction t (db.begin ());

  e = db.load<employee> (id1);
  e->temporary (false);
  p = e;
  db.update (p); // Updates employee.

  t.commit ();
}

// Erase.
//
{
  shared_ptr<person> p;

  transaction t (db.begin ());
  p = db.load<person> (id1); // Loads employee.
```

```
  db.erase (p);               // Erases employee.
  db.erase<person> (id2);     // Erases contractor.
  t.commit ();
}
```

The table-per-difference mapping, as supported by ODB, requires two extra columns, in addition to those corresponding to the data members. The first, called *discriminator*, is added to the table corresponding to the root class of the hierarchy. This column is used to determine the dynamic type of each object. The second column is added to tables corresponding to the derived classes and contains the object id. This column is used to form a foreign key constraint referencing the root class table.

When querying the database for polymorphic objects, it is possible to obtain the discriminator value without instantiating the object. For example:

```
using query = odb::query<person>;
using result = odb::result<person>;

transaction t (db.begin ());

result r (db.query<person> (query::last == "Doe"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  std::string d (i.discriminator ());
  ...
}

t.commit ();
```

In the current implementation, ODB has limited support for customizing names, types, and values of the extra columns. Currently, the discriminator column is always called `typeid` and contains a namespace-qualified class name (for example, `"employee"` or `"hr::employee"`). The id column in the derived class table has the same name as the object id column in the root class table. Future versions of ODB will add support for customizing these extra columns.

The sample database schema for the above polymorphic hierarchy is shown below.

```
CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
  typeid VARCHAR(128) NOT NULL,
  first TEXT NOT NULL,
  last TEXT NOT NULL);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  temporary TINYINT(1) NOT NULL,
```

```
  CONSTRAINT employee_id_fk
    FOREIGN KEY (id)
    REFERENCES person (id)
    ON DELETE CASCADE);

CREATE TABLE contractor (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  email TEXT NOT NULL,

  CONSTRAINT contractor_id_fk
    FOREIGN KEY (id)
    REFERENCES person (id)
    ON DELETE CASCADE);
```

The complete version of the code presented in this section is available in the `inheritance/polymorphism` example in the `odb-examples` package.

8.2.1 Performance and Limitations

A database operation on a non-polymorphic object normally translates to a single database statement execution (objects with containers and eager object pointers can be the exception). Because polymorphic objects have their data members stored in multiple tables, some database operations on such objects may result in multiple database statements being executed while others may require more complex statements. There is also some functionality that is not available to polymorphic objects.

The first part of this section discusses the performance implications to keep in mind when designing and working with polymorphic hierarchies. The second part talks about limitations of polymorphic objects.

The most important aspect of a polymorphic hierarchy that affects database performance is its depth. The distance between the root of the hierarchy and the derived class translates directly to the number of database statements that will have to be executed in order to persist, update, or erase this derived class. It also translates directly to the number of SQL `JOIN` clauses that will be needed to load or query the database for this derived class. As a result, to achieve best performance, we should try to keep our polymorphic hierarchies as flat as possible.

When loading an object or querying the database for objects, ODB will need to execute two statements if this object's static and dynamic types are different but only one statement if they are the same. This example will help illustrate the difference:

```
unsigned long long id;

{
  employee e (...);

  transaction t (db.begin ());
```

```
  id = db.persist (e);
  t.commit ();
}

{
  shared_ptr<person> p;

  transaction t (db.begin ());
  p = db.load<person> (id);   // Requires two statement.
  p = db.load<employee> (id); // Requires only one statement.
  t.commit ();
}
```

As a result, we should try to load and query using the most derived class possible.

Finally, for polymorphic objects, erasing via the object instance is faster than erasing via its object id. In the former case the object's dynamic type can be determined locally in the application while in the latter case an extra statement has to be executed to achieve the same result. For example:

```
shared_ptr<person> p = ...;

transaction t (db.begin ());
db.erase<person> (p.id ()); // Slower (executes extra statement).
db.erase (p);               // Faster.
t.commit ();
```

Polymorphic objects can use all the mechanisms that are available to ordinary objects. These include containers (Chapter 5, "Containers"), object relationships, including to polymorphic objects (Chapter 6, "Relationships"), views (Chapter 10, "Views"), session (Chapter 11, "Session"), and optimistic concurrency (Chapter 12, "Optimistic Concurrency"). There are, however, a few limitations, mainly due to the underlying use of SQL to access the data.

When a polymorphic object is "joined" in a view, and the join condition (either in the form of an object pointer or a custom condition) comes from the object itself (as opposed to one of the objects joined previously), then this condition must only use data members from the derived class. For example, consider the following polymorphic object hierarchy and a view:

```
#pragma db object polymorphic
class employee
{
  ...
};

#pragma db object
class permanent_employee: public employee
{
  ...
```

```
};

#pragma db object
class temporary_employee: public employee
{
  ...

  shared_ptr<permanent_employee> manager_;
};

#pragma db object
class contractor: public temporary_employee
{
  shared_ptr<permanent_employee> manager_;
};

#pragma db view object(permanent_employee) \
               object(contractor: contractor::manager_)
struct contractor_manager
{
  ...
};
```

This view will not function correctly because the join condition (`manager_`) comes from the base class (`temporary_employee`) instead of the derived (`contractor`). The reason for this limitation is the `JOIN` clause order in the underlying SQL `SELECT` statement. In the view presented above, the table corresponding to the base class (`temporary_employee`) will have to be joined first which will result in this view matching both the `temporary_employee` and `contractor` objects instead of just `contractor`. It is usually possible to resolve this issue by reordering the objects in the view. Our example, for instance, can be fixed by swapping the two objects:

```
#pragma db view object(contractor) \
               object(permanent_employee: contractor::manager_)
struct contractor_manager
{
  ...
};
```

The `erase_query()` database function (Section 3.11, "Deleting Persistent Objects") also has limited functionality when used on polymorphic objects. Because many database implementations do not support `JOIN` clauses in the SQL `DELETE` statement, only data members from the derived class being erased can be used in the query condition. For example:

```
using query = odb::query<employee>;

transaction t (db.begin ());
db.erase_query<employee> (query::permanent);     // Ok.
db.erase_query<employee> (query::last == "Doe"); // Error.
t.commit ();
```

8.3 Mixed Inheritance

It is possible to mix the reuse and polymorphism inheritance styles in the same hierarchy. In this case, the reuse inheritance must be used for the "bottom" (base) part of the hierarchy while the polymorphism inheritance — for the "top" (derived) part. For example:

```
#pragma db object
class person
{
  ...
};

#pragma db object polymorphic
class employee: public person // Reuse inheritance.
{
  ...
};

#pragma db object
class temporary_employee: public employee // Polymorphism inheritance.
{
  ...
};

#pragma db object
class permanent_employee: public employee // Polymorphism inheritance.
{
  ...
};
```

9 Sections

ODB sections are an optimization mechanism that allows us to partition data members of a persistent class into groups that can be separately loaded and/or updated. This can be useful, for example, if an object contains expensive to load or update data members (such as `BLOB`s or containers) and that are accessed or modified infrequently. For example:

```
#include <odb/section.hxx>

#pragma db object
class person
{
  ...

  #pragma db load(lazy) update(manual)
  odb::section keys_;

  #pragma db section(keys_) type("BLOB")
  char public_key_[1024];

  #pragma db section(keys_) type("BLOB")
  char private_key_[1024];
};

transaction t (db.begin ());

unique_ptr<person> p (db.load<person> (...)); // Keys are not loaded.

if (need_keys)
{
  db.load (*p, p->keys_); // Load keys.
  ...
}

db.update (*p); // Keys are not updated.

if (update_keys)
{
  ...
  db.update (*p, p->keys_); // Update keys.
}

t.commit ();
```

A complete example that shows how to use sections is available in the `section` directory in the `odb-examples` package.

Why do we need to group data members into sections? Why can't each data member be loaded and updated independently if and when necessary? The reason for this requirement is that loading or updating a group of data members with a single database statement is significantly more efficient than loading or updating each data member with a separate statement. Because ODB prepares and caches statements used to load and update persistent objects, generating a custom statement for a specific set of data members that need to be loaded or updated together is not a viable approach either. To resolve this, ODB allows us to group data members that are often updated and/or loaded together into sections. To achieve the best performance, we should aim to find a balance between having too many sections with too few data members and too few sections with too many data members. We can use the access and modification patterns of our application as a base for this decision.

To add a new section to a persistent class we declare a new data member of the `odb::section` type. At this point we also need to specify the loading and updating behavior of this section with the `db load` and `db update` pragmas, respectively.

The loading behavior of a section can be either `eager` or `lazy`. An eager-loaded section is always loaded as part of the object load. A lazy-loaded section is not loaded as part of the object load and has to be explicitly loaded with the `database::load()` function (discussed below) if and when necessary.

The updating behavior of a section can be `always`, `change`, or `manual`. An always-updated section is always updated as part of the object update, provided it has been loaded. A change-updated section is only updated as part of the object update if it has been loaded and marked as changed. A manually-updated section is never updated as part of the object update and has to be explicitly updated with the `database::update()` function (discussed below) if and when necessary.

If no loading behavior is specified explicitly, then an eager-loaded section is assumed. Similarly, if no updating behavior is specified, then an always-updated section is assumed. An eager-loaded, always-updated section is pointless and therefore illegal. Only persistent classes with an object id can have sections.

To specify that a data member belongs to a section we use the `db section` pragma with the section's member name as its single argument. Except for special data members such as the object id and optimistic concurrency version, any direct, non-transient member of a persistent class can belong to a section, including composite values, containers, and pointers to objects. For example:

```
#pragma db value
class text
{
  std::string data;
  std::string lang;
};
```

```
#pragma db object
class person
{
  ...

  #pragma db load(lazy)
  odb::section extras_;

  #pragma db section(extras_)
  text bio_;

  #pragma db section(extras_)
  std::vector<std::string> nicknames_;

  #pragma db section(extras_)
  std::shared_ptr<person> emergency_contact_;
};
```

An empty section is pointless and therefore illegal, except in abstract or polymorphic classes where data members can be added to a section by derived classes (see Section 9.1, "Sections and Inheritance").

The `odb::section` class is defined in the `<odb/section.hxx>` header file and has the following interface:

```
namespace odb
{
  class section
  {
  public:
    // Load state.
    //
    bool
    loaded () const;

    void
    unload ();

    void
    load ();

    // Change state.
    //
    bool
    changed () const;

    void
    change ();
```

```
    // User data.
    //
    unsigned char
    user_data () const;

    void
    user_data (unsigned char);
  };
}
```

The `loaded()` accessor can be used to determine whether a section is already loaded. The `unload()` modifier marks a loaded section as not loaded. This, for example, can be useful if you don't want the section to be reloaded during the object reload. The `load()` modifier marks an unloaded section as loaded without actually loading any of its data members. This, for example, can be useful if you don't want to load the old state before overwriting it with `update()`.

The `changed()` accessor can be used to query the section's change state. The `change()` modifier marks the section as changed. It is valid to call this modifier for an unloaded (or transient) section, however, the state will be reset back to unchanged once the section (or object) is loaded. The change state is only relevant to sections with change-updated behavior and is ignored for all other sections.

The size of the section class is one byte with four bits available to store a custom state via the `user_data()` accessor and modifier.

The `odb::database` class provides special versions of the `load()` and `update()` functions that allow us to load and update sections of a persistent class. Their signatures are as follows:

```
template <typename T>
void
load (T& object, section&);

template <typename T>
void
update (const T& object, const section&);
```

Before calling the section `load()` function, the object itself must already be loaded. If the section is already loaded, then the call to `load()` will reload its data members. It is illegal to explicitly load an eager-loaded section.

Before calling the section `update()` function, the section (and therefore the object) must be in the loaded state. If the section is not loaded, the `odb::section_not_loaded` exception is thrown. The section `update()` function does not check but does clear the section's change state. In other words, section `update()` will always update section data members in the database and clear the change flag. Note also that any section, that is, always-, change-, or manu-

ally-updated, can be explicitly updated with this function.

Both section `load()` and `update()`, just like the rest of the database operations, must be performed within a transaction. Notice also that both `load()` and `update()` expect a reference to the section as their second argument. This reference must refer to the data member in the object passed as the first argument. If instead it refers to some other instance of the `section` class, for example, a local copy or a temporary, then the `odb::section_not_in_object` exception is thrown. For example:

```
#pragma db object
class person
{
public:
  ...

  odb::section
  keys () const {return keys_;}

private:
  odb::section keys_;

  ...
};

unique_ptr<person> p (db.load<person> (...));

section s (p->keys ());
db.load (*p, s);            // Throw section_not_in_object, copy.

db.update (*p, p->keys ()); // Throw section_not_in_object, copy.
```

At first glance it may seem more appropriate to make the `section` class non-copyable in order to prevent such errors from happening. However, it is perfectly reasonable to expect to be able to copy (or assign) sections as part of the object copying (or assignment). As a result, sections are left copyable and copy-assignable, however, this functionality should not be used in accessors or modifiers. Instead, section accessors and modifiers should always be by-reference. Here is how we can fix our previous example:

```
#pragma db object
class person
{
public:
  ...

  const odb::section&
  keys () const {return keys_;}

  odb::section&
  keys () {return keys_;}
```

```
private:
  odb::section keys_;

  ...
};

unique_ptr<person> p (db.load<person> (...));

section& s (p->keys ());
db.load (*p, s);            // Ok, reference.

db.update (*p, p->keys ()); // Ok, reference.
```

Several other database operations affect sections. The state of a section in a transient object is undefined. That is, before the call to object `persist()` or `load()` functions, or after the call to object `erase()` function, the values returned by the `section::loaded()` and `section::changed()` accessors are undefined.

After the call to `persist()`, all sections, including eager-loaded ones, are marked as loaded and unchanged. If instead we are loading an object with the `load()` call or as a result of a query, then eager-loaded sections are loaded and marked as loaded and unchanged while lazy-loaded ones are marked as unloaded. If a lazy-loaded section is later loaded with the section `load()` call, then it is marked as loaded and unchanged.

When we update an object with the `update()` call, manually-updated sections are ignored while always-updated sections are updated if they are loaded. Change-updated sections are only updated if they are both loaded and marked as changed. After the update, such sections are reset to the unchanged state. When we reload an object with the `reload()` call, sections that were loaded are automatically reloaded and reset to the unchanged state.

To further illustrate the state transitions of a section, consider this example:

```
#pragma db object
class person
{
  ...

  #pragma db load(lazy) update(change)
  odb::section keys_;

  ...
};

transaction t (db.begin ());

person p ("John", "Doe"); // Section state is undefined (transient).
```

```
db.persist (p);           // Section state: loaded, unchanged.

unique_ptr<person> l (
  db.load<person> (...)); // Section state: unloaded, unchanged.

db.update (*l);           // Section not updated since not loaded.
db.update (p);            // Section not updated since not changed.

p.keys_.change ();        // Section state: loaded, changed.
db.update (p);            // Section updated, state: loaded, unchanged.

db.update (*l, l->keys_); // Throw section_not_loaded.
db.update (p, p.keys_);   // Section updated even though not changed.

db.reload (*l);           // Section not reloaded since not loaded.
db.reload (p);            // Section reloaded, state: loaded, unchanged.

db.load (*l, l->keys_);   // Section loaded, state: loaded, unchanged.
db.load (p, p.keys_);     // Section reloaded, state: loaded, unchanged.

db.erase (p);             // Section state is undefined (transient).

t.commit ();
```

When using change-updated behavior, it is our responsibility to mark the section as changed when any of the data members belonging to this section is modified. A natural place to mark the section as changed is the modifiers for section data members, for example:

```
#pragma db object
class person
{
  ...

  using key_type = std::array<char, 1024>;

  const key_type&
  public_key () const {return public_key_;}

  void
  public_key (const key_type& k)
  {
    public_key_ = k;
    keys_.change ();
  }

  const key_type&
  private_key () const {return private_key_;}

  void
  private_key (const key_type& k)
  {
```

```
    private_key_ = k;
    keys_.change ();
  }

private:
  #pragma db load(lazy) update(change)
  odb::section keys_;

  #pragma db section(keys_) type("BLOB")
  key_type public_key_;

  #pragma db section(keys_) type("BLOB")
  key_type private_key_;

  ...
};
```

One interesting aspect of change-updated sections is what happens when a transaction that performed an object or section update is later rolled back. In this case, while the change state of a section has been reset (after update), actual changes were not committed to the database. Change-updated sections handle this case by automatically registering a rollback callback and then, if it is called, restoring the original change state. The following code illustrates this semantics (continuing with the previous example):

```
unique_ptr<person> p;

try
{
  transaction t (db.begin ());
  p = db.load<person> (...);
  db.load (*p, p->keys_);

  p->private_key (new_key); // The section is marked changed.
  db.update (*p);          // The section is reset to unchanged.

  throw failed ();         // Triggers rollback.
  t.commit ();
}
catch (const failed&)
{
  // The section is restored back to changed.
}
```

9.1 Sections and Inheritance

With both reuse and polymorphism inheritance (Chapter 8, "Inheritance") it is possible to add new sections to derived classes. It is also possible to add data members from derived classes to sections declared in the base. For example:

```
#pragma db object polymorphic
class person
{
  ...

  virtual void
  print ();

  #pragma db load(lazy)
  odb::section print_;

  #pragma db section(print_)
  std::string bio_;
};

#pragma db object
class employee: public person
{
  ...

  virtual void
  print ();

  #pragma db section(print_)
  std::vector<std::string> employment_history_;
};

transaction t (db.begin ());

unique_ptr<person> p (db.load<person> (...)); // Person or employee.
db.load (*p, p->print_); // Load data members needed for print.
p->print ();

t.commit ();
```

When data members of a section are spread over several classes in a reuse inheritance hierarchy, both section load and update are performed with a single database statement. In contrast, with polymorphism inheritance, section load is performed with a single statement while update requires a separate statement for each class that adds to the section.

Note also that in polymorphism inheritance the section-to-object association is static. Or, in other words, you can load a section via an object only if its static type actually contains this section. The following example will help illustrate this point further:

```
#pragma db object polymorphic
class person
{
  ...
};
```

```
#pragma db object
class employee: public person
{
  ...

  #pragma db load(lazy)
  odb::section extras_;

  ...
};

#pragma db object
class manager: public employee
{
  ...
};

unique_ptr<manager> m (db.load<manager> (...));

person& p (*m);
employee& e (*m);
section& s (m->extras_);

db.load (p, s); // Error: extras_ is not in person.
db.load (e, s); // Ok: extras_ is in employee.
```

9.2 Sections and Optimistic Concurrency

When sections are used in a class with the optimistic concurrency model (Chapter 12, "Optimistic Concurrency"), both section update and load operations compare the object version to that in the database and throw the `odb::object_changed` exception if they do not match. In addition, the section update operation increments the version to indicate that the object state has changed. For example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version
  unsigned long long version_;

  #pragma db load(lazy)
  odb::section extras_;

  #pragma db section(extras_)
  std::string bio_;
};
```

```
unique_ptr<person> p;

{
  transaction t (db.begin ());
  p = db.load<person> (...);
  t.commit ();
}

{
  transaction t (db.begin ());

  try
  {
    db.load (*p, p->extras_); // Throws if object state has changed.
  }
  catch (const object_changed&)
  {
    db.reload (*p);
    db.load (*p, p->extras_); // Cannot fail.
  }

  t.commit ();
}
```

Note also that if an object update triggers one or more section updates, then each such update will increment the object version. As a result, an update of an object that contains sections may result in a version increment by more than one.

When sections are used together with optimistic concurrency and inheritance, an extra step may be required to enable this functionality. If you plan to add new sections to derived classes, then the root class of the hierarchy (the one that declares the version data member) must be declared as sectionable with the `db sectionable` pragma. For example:

```
#pragma db object polymorphic sectionable
class person
{
  ...

  #pragma db version
  unsigned long long version_;
};

#pragma db object
class employee: public person
{
  ...

  #pragma db load(lazy)
  odb::section extras_;
```

```
  #pragma db section(extras_)
  std::vector<std::string> employment_history_;
};
```

This requirement has to do with the need to generate extra version increment code in the root class that will be used by sections added in the derived classes. If you forget to declare the root class as sectionable and later add a section to one of the derived classes, the ODB compiler will issue diagnostics.

9.3 Sections and Lazy Pointers

If a lazy pointer (Section 6.4, "Lazy Pointers") belongs to a lazy-loaded section, then we end up with two levels of lazy loading. Specifically, when the section is loaded, the lazy pointer is initialized with the object id but the object itself is not loaded. For example:

```
#pragma db object
class employee
{
  ...

  #pragma db load(lazy)
  odb::section extras_;

  #pragma db section(extras_)
  odb::lazy_shared_ptr<employer> employer_;
};

transaction t (db.begin ());

unique_ptr<employee> e (db.load<employee> (...)); // employer_ is NULL.

db.load (*e, e->extras_); // employer_ contains valid employer id.

e->employer_.load (); // employer_ points to employer object.

t.commit ();
```

9.4 Sections and Change-Tracking Containers

If a change-tracking container (Section 5.4, "Change-Tracking Containers") belongs to a change-updated section, then prior to an object update ODB will check if the container has been changed and if so, automatically mark the section as changed. For example:

```
#pragma db object
class person
{
  ...
```

```
  #pragma db load(lazy) update(change)
  odb::section extras_;

  #pragma db section(extras_)
  odb::vector<std::string> nicknames_;
};

transaction t (db.begin ());

unique_ptr<person> p (db.load<person> (...));
db.load (*p, p->extras_);

p->nicknames_.push_back ("JD");

db.update (*p); // Section is automatically updated even
                // though it was not marked as changed.
t.commit ();
```

10 Views

An ODB view is a C++ `class` or `struct` type that embodies a light-weight, read-only projection of one or more persistent objects or database tables or the result of a native SQL query execution.

Some of the common applications of views include loading a subset of data members from objects or columns from database tables, executing and handling results of arbitrary SQL queries, including aggregate queries and stored procedure calls, as well as joining multiple objects and/or database tables using object relationships or custom join conditions.

Many relational databases also define the concept of views. Note, however, that ODB views are not mapped to database views. Rather, by default, an ODB view is mapped to an SQL `SELECT` query. However, if desired, it is easy to create an ODB view that is based on a database view.

Usually, views are defined in terms of other persistent entities, such as persistent objects, database tables, sequences, etc. Therefore, before we can examine our first view, we need to define a few persistent objects and a database table. We will use this model in examples throughout this chapter. Here we assume that you are familiar with ODB object relationship support (Chapter 6, "Relationships").

```
#pragma db object
class country
{
  ...

  #pragma db id
  std::string code_; // ISO 2-letter country code.

  std::string name_;
};

#pragma db object
class employer
{
  ...

  #pragma db id
  unsigned long long id_;

  std::string name_;
};

#pragma db object
class employee
{
  ...
```

```
  #pragma db id
  unsigned long long id_;

  std::string first_;
  std::string last_;

  unsigned short age_;

  shared_ptr<country> residence_;
  shared_ptr<country> nationality_;

  shared_ptr<employer> employed_by_;
};
```

Besides these objects, we also have the legacy `employee_extra` table that is not mapped to any persistent class. It has the following definition:

```
CREATE TABLE employee_extra(
  employee_id INTEGER NOT NULL,
  vacation_days INTEGER NOT NULL,
  previous_employer_id INTEGER)
```

The above persistent objects and database table as well as many of the views shown in this chapter are based on the `view` example which can be found in the `odb-examples` package of the ODB distribution.

To declare a view we use the `db view` pragma, for example:

```
#pragma db view object(employee)
struct employee_name
{
  std::string first;
  std::string last;
};
```

The above example shows one of the simplest views that we can create. It has a single associated object (`employee`) and its purpose is to extract the employee's first and last names without loading any other data, such as the referenced `country` and `employer` objects.

Views use the same query facility (Chapter 4, "Querying the Database") as persistent objects. Because support for queries is optional and views cannot be used without this support, you need to compile any header that defines a view with the `--generate-query` ODB compiler option.

To query the database for a view we use the `database::query()`, `database::query_one()`, or `database::query_value()` functions in exactly the same way as we would use them to query the database for an object. For example, the following code fragment shows how we can find the names of all the employees that are younger than 31:

```
using query = odb::query<employee_name>;
using result = odb::result<employee_name>;

transaction t (db.begin ());

result r (db.query<employee_name> (query::age < 31));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  const employee_name& en (*i);
  cout << en.first << " " << en.last << endl;
}

t.commit ();
```

A view can be defined as a projection of one or more objects, one or more tables, a combination of objects and tables, or it can be the result of a custom SQL query. The following sections discuss each of these kinds of view in more detail.

10.1 Object Views

To associate one or more objects with a view we use the `db object` pragma (Section 14.2.1, "`object`"). We have already seen a simple, single-object view in the introduction to this chapter. To associate the second and subsequent objects we repeat the `db object` pragma for each additional object, for example:

```
#pragma db view object(employee) object(employer)
struct employee_employer
{
  std::string first;
  std::string last;
  std::string name;
};
```

The complete syntax of the `db object` pragma is shown below:

`object(`*`name`* `[`**`=`** *`alias`*`]` `[`*`join-type`*`]` `[:` *`join-condition`*`]`**`)`**

The *name* part is a potentially qualified persistent class name that has been defined previously. The optional *alias* part gives this object an alias. If provided, the alias is used in several contexts instead of the object's unqualified name. We will discuss aliases further as we cover each of these contexts below. The optional *join-type* part specifies the way this object is associated. It can be `left`, `right`, `full`, `inner`, and `cross` with `left` being the default. Finally, the optional *join-condition* part provides the criteria which should be used to associate this object with any of the previously associated objects or, as we will see in Section 10.4, "Mixed Views", tables. Note that while the first associated object can have an alias, it cannot have a join type or condition.

For each subsequent associated object the ODB compiler needs a join condition and there are several ways to specify it. The easiest way is to omit it altogether and let the ODB compiler try to come up with a join condition automatically. To do this the ODB compiler will examine each previously associated object for object relationships (Chapter 6, "Relationships") that may exist between these objects and the object being associated. If such a relationship exists and is unambiguous, that is there is only one such relationship, then the ODB compiler will automatically use it to come up with the join condition for this object. This is exactly what happens in the previous example: there is a single relationship (`employee::employed_by`) between the `employee` and `employer` objects.

On the other hand, consider this view:

```
#pragma db view object(employee) object(country)
struct employee_residence
{
  std::string first;
  std::string last;
  std::string name;
};
```

While there is a relationship between `country` and `employee`, it is ambiguous. It can be `employee::residence_` (which is what we want) or it can be `employee::nationality_` (which we don't want). As result, when compiling the above view, the ODB compiler will issue an error indicating an ambiguous object relationship. To resolve this ambiguity, we can explicitly specify the object relationship that should be used to create the join condition as the name of the corresponding data member. Here is how we can fix the `employee_residence` view:

```
#pragma db view object(employee) object(country: employee::residence_)
struct employee_residence
{
  std::string first;
  std::string last;
  std::string name;
};
```

It is possible to associate the same object with a single view more than once using different join conditions. However, in this case, we have to use aliases to assign different names for each association. For example:

```
#pragma db view object(employee) \
  object(country = res_country: employee::residence_) \
  object(country = nat_country: employee::nationality_)
struct employee_country
{
  ...
};
```

Note that correctly defining data members in this view requires the use of a mechanism that we haven't yet covered. We will see how to do this shortly.

If we assign an alias to an object and refer to a data member of this object in one of the join conditions, we have to use the unqualified alias name instead of the potentially qualified object name. For example:

```
#pragma db view object(employee = ee) object(country: ee::residence_)
struct employee_residence
{
  ...
};
```

The last way to specify a join condition is to provide a custom query expression. This method is primarily useful if you would like to associate an object using a condition that does not involve an object relationship. Consider, for example, a modified `employee` object from the beginning of the chapter with an added country of birth member. For one reason or another we have decided not to use a relationship to the `country` object, as we have done with residence and nationality.

```
#pragma db object
class employee
{
  ...

  std::string birth_place_; // Country name.
};
```

If we now want to create a view that returns the birth country code for an employee, then we have to use a custom join condition when associating the `country` object. For example:

```
#pragma db view object(employee) \
  object(country: employee::birth_place_ == country::name_)
struct employee_birth_code
{
  std::string first;
  std::string last;
  std::string code;
};
```

The syntax of the query expression in custom join conditions is the same as in the query facility used to query the database for objects (Chapter 4, "Querying the Database") except that for query members, instead of using `odb::query<object>::member` names, we refer directly to object members.

Looking at the views we have defined so far, you may be wondering how the ODB compiler knows which view data members correspond to which object data members. While the names are similar, they are not exactly the same, for example `employee_name::first` and

`employee::first_`.

As with join conditions, when it comes to associating data members, the ODB compiler tries to do this automatically. It first searches all the associated objects for an exact name match. If no match is found, then the ODB compiler compares the so-called public names. A public name of a member is obtained by removing the common member name decorations, such as leading and trailing underscores, the `m_` prefix, etc. In both of these searches the ODB compiler also makes sure that the types of the two members are the same or compatible.

If one of the above searches returned a match and it is unambiguous, that is there is only one match, then the ODB compiler will automatically associate the two members. On the other hand, if no match is found or the match is ambiguous, the ODB compiler will issue an error. To associate two differently-named members or to resolve an ambiguity, we can explicitly specify the member association using the `db column` pragma (Section 14.4.9, "`column`"). For example:

```
#pragma db view object(employee) object(employer)
struct employee_employer
{
  std::string first;
  std::string last;

  #pragma db column(employer::name_)
  std::string employer_name;
};
```

If an object data member specifies the SQL type with the `db type` pragma (Section 14.4.3, "`type`"), then this type is also used for the associated view data members.

Note also that similar to join conditions, if we assign an alias to an object and refer to a data member of this object in one of the `db column` pragmas, then we have to use the unqualified alias name instead of the potentially qualified object name. For example:

```
#pragma db view object(employee) \
  object(country = res_country: employee::residence_) \
  object(country = nat_country: employee::nationality_)
struct employee_country
{
  std::string first;
  std::string last;

  #pragma db column(res_country::name_)
  std::string res_country_name;

  #pragma db column(nat_country::name_)
  std::string nat_country_name;
};
```

Besides specifying just the object member, we can also specify a +-*expression* in the `db column` pragma. A +-expression consists of string literals and object member references connected using the + operator. It is primarily useful for defining aggregate views based on SQL aggregate functions, for example:

```
#pragma db view object(employee)
struct employee_count
{
  #pragma db column("count(" + employee::id_ + ")")
  std::size_t count;
};
```

When querying the database for a view, we may want to provide additional query criteria based on the objects associated with this view. To support this a view defines query members for all the associated objects which allows us to refer to such objects' members using the `odb::query<view>::member` expressions. This is similar to how we can refer to object members using the `odb::query<object>::member` expressions when querying the database for an object. For example:

```
using query = odb::query<employee_count>;

transaction t (db.begin ());

// Find the number of employees with the Doe last name. Result of this
// aggregate query contains only one element so use the query_value()
// shortcut function.
//
employee_count ec (
  db.query_value<employee_count> (query::last == "Doe"));

cout << ec.count << endl;

t.commit ();
```

In the above query we used the last name data member from the associated `employee` object to only count employees with the specific name.

When a view has only one associated object, the query members corresponding to this object are defined directly in the `odb::query<view>` scope. For instance, in the above example, we referred to the last name member as `odb::query<employee_count>::last`. However, if a view has multiple associated objects, then query members corresponding to each such object are defined in a nested scope named after the object. As an example, consider the `employee_employer` view again:

```
#pragma db view object(employee) object(employer)
struct employee_employer
{
  std::string first;
  std::string last;

  #pragma db column(employer::name_)
  std::string employer_name;
};
```

Now, to refer to the last name data member from the `employee` object we use the `odb::query<...>::employee::last` expression. Similarly, to refer to the employer name, we use the `odb::query<...>::employer::name` expression. For example:

```
using result = odb::result<employee_employer>;
using query = odb::query<employee_employer>;

transaction t (db.begin ());

result r (db.query<employee_employer> (
  query::employee::last == "Doe" &&
  query::employer::name == "Simple Tech Ltd"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cout << i->first << " " << i->last <<  " " << i->employer_name << endl;

t.commit ();
```

If we assign an alias to an object, then this alias is used to name the query members scope instead of the object name. As an example, consider the `employee_country` view again:

```
#pragma db view object(employee) \
  object(country = res_country: employee::residence_) \
  object(country = nat_country: employee::nationality_)
struct employee_country
{
  ...
};
```

And a query which returns all the employees that have the same country of residence and nationality:

```
using query = odb::query<employee_country>;
using result = odb::result<employee_country>;

transaction t (db.begin ());

result r (db.query<employee_country> (
  query::res_country::name == query::nat_country::name));
```

```
for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cout << i->first << " " << i->last << " " << i->res_country_name << endl;

t.commit ();
```

Note also that unlike object query members, view query members do no support referencing members in related objects. For example, the following query is invalid:

```
using query = odb::query<employee_name>;
using result = odb::result<employee_name>;

transaction t (db.begin ());

result r (db.query<employee_name> (
  query::employed_by->name == "Simple Tech Ltd"));

t.commit ();
```

To get this behavior, we would instead need to associate the `employer` object with this view and then use the `query::employer::name` expression instead of `query::employed_by->name`.

As we have discussed above, if specified, an object alias is used instead of the object name in the join condition, data member references in the `db column` pragma, as well as to name the query members scope. The object alias is also used as a table name alias in the underlying `SELECT` statement generated by the ODB compiler. Normally, you would not use the table alias directly with object views. However, if for some reason you need to refer to a table column directly, for example, as part of a native query expression, and you need to qualify the column with the table, then you will need to use the table alias instead.

10.2 Object Loading Views

A special variant of object views is object loading views. Object loading views allow us to load one or more complete objects instead of, or in addition to, a subset of data member. While we can often achieve the same end result by calling `database::load()`, using a view has several advantages.

If we need to load multiple objects, then using a view allows us to do this with a single `SELECT` statement execution instead of one for each object that would be necessary in case of `load()`. A view can also be useful for loading only a single object if the query criterion that we would like to use involves other, potentially unrelated, objects. We will examine concrete examples of these and other scenarios in the rest of this section.

To load a complete object as part of a view we use a data member of the pointer to object type, just like for object relationships (Chapter 6, "Relationships"). As an example, here is how we can load both the `employee` and `employer` objects from the previous section with a single statement:

```
#pragma db view object(employee) object(employer)
struct employee_employer
{
  shared_ptr<employee> ee;
  shared_ptr<employer> er;
};
```

We use an object loading view just like any other view. In the result of a query, as we would expect, the pointer data members point to the loaded objects. For example:

```
using query = odb::query<employee_employer>;

transaction t (db.begin ());

for (const employee_employer& r:
       db.query<employee_employer> (query::employee::age < 31))
{
  cout << r.ee->age () << " " << r.er->name () << endl;
}

t.commit ();
```

As another example, consider a query that loads the `employer` objects using some condition based on its employees. For instance, we want to find all the employers that employ people over 65 years old. We can use this object loading view to implement such a query (notice the `distinct` result modifier discussed later in Section 10.5, "View Query Conditions"):

```
#pragma db view object(employer) object(employee) query(distinct)
struct employer_view
{
  shared_ptr<employer> er;
};
```

And this is how we can use this view to find all the employers that employ seniors:

```
using query = odb::query<employer_view>;

db.query<employer_view> (query::employee::age > 65)
```

We can even use object loading views to load completely unrelated (from the ODB object relationships point of view) objects. For example, the following view will load all the employers that are named the same as a country (notice the `inner` join type):

```
#pragma db view object(employer) \
  object(country inner: employer::name == country::name)
struct employer_named_country
{
  shared_ptr<employer> e;
  shared_ptr<country> c;
};
```

An object loading view can contain ordinary data members in addition to object pointers. For example, if we are only interested in the country code in the above view, then we can reimplement it like this:

```
#pragma db view object(employer) \
  object(country inner: employer::name == country::name)
struct employer_named_country
{
  shared_ptr<employer> e;
  std::string code;
};
```

Object loading views also have a few rules and restrictions. Firstly, the pointed-to object in the data member must be associated with the view. Furthermore, if the associated object has an alias, then the data member name must be the same as the alias (more precisely, the public name derived from the data member must match the alias; which means we can use normal data member decorations such as trailing underscores, etc., see the previous section for more information on public names). The following view illustrates the use of aliases as data member names:

```
#pragma db view object(employee)                \
  object(country = res: employee::residence_)   \
  object(country = nat: employee::nationality_)
struct employee_country
{
  shared_ptr<country> res;
  shared_ptr<country> nat_;
};
```

Finally, the object pointers must be direct data members of the view. Using, for example, a composite value that contains pointers as a view data member is not supported. Note also that depending on the join type you are using, some of the resulting pointers might be `NULL`.

Up until now we have consistently used `shared_ptr` as an object pointer in our views. Can we use other pointers, such as `unique_ptr` or raw pointers? To answer this question we first need to discuss what happens with object pointers that may be inside objects that a view loads. As a concrete example, let us revisit the `employee_employer` view from the beginning of this section:

```
#pragma db view object(employee) object(employer)
struct employee_employer
{
  shared_ptr<employee> ee;
  shared_ptr<employer> er;
};
```

This view loads two objects: `employee` and `employer`. The `employee` object, however, also contains a pointer to `employer` (see the `employed_by_` data member). In fact, this is the same object that the view loads since `employer` is associated with the view using this same relationship (ODB automatically uses it since it is the only one). The correct result of loading such a view is then clear: both `er` and `er->employed_by_` must point to (or share) the same instance.

Just like object loading via the `database` class functions, views achieve this correct behavior of only loading a single instance of the same object with the help of session's object cache (Chapter 11, "Session"). In fact, object loading views enforce this by throwing the `session_required` exception if there is no current session and the view loads an object that is also indirectly loaded by one of the other objects. The ODB compiler will also issue diagnostics if such an object has session support disabled (Section 14.1.10, "`session`").

With this understanding we can now provide the correct implementation of our transaction that uses the `employee_employer` view:

```
using query = odb::query<employee_employer>;

transaction t (db.begin ());
odb::session s;

for (const employee_employer& r:
       db.query<employee_employer> (query::employee::age < 31))
{
  assert (r.ee->employed_by_ == r.er);
  cout << r.ee->age () << " " << r.er->name () << endl;
}

t.commit ();
```

It might seem logical, then, to always load all the objects from all the eager relationships with the view. After all, this will lead to them all being loaded with a single statement. While this is theoretically true, the reality is slightly more nuanced. If there is a high probability of the object already have been loaded and sitting in the cache, then not loading the object as part of the view (and therefore not fetching all its data from the database) might result in better performance.

Now we can also answer the question about which pointers we can use in object loading views. From the above discussion it should be clear that if an object that we are loading is also part of a relationship inside another object that we are loading, then we should use some form of a shared ownership pointer. If, however, there are no relationships involved, as is the case, for example, in our `employer_named_country` and `employee_country` views above, then we can use a unique ownership pointer such as `unique_ptr`.

Note also that your choice of a pointer type can be limited by the "official" object pointer type assigned to the object (Section 3.3, "Object and View Pointers"). For example, if the object pointer type is `shared_ptr`, you will not be able to use `unique_ptr` to load such an object into a view since initializing `unique_ptr` from `shared_ptr` would be a mistake.

Unless you want to perform your own object cleanup, raw object pointers in views are not particularly useful. They do have one special semantics, however: If a raw pointer is used as a view member, then, before creating a new instance, the implementation will check if the member is `NULL`. If it is not, then it is assumed to point to an existing instance and the implementation will load the data into it instead of creating a new one. The primary use of this special functionality is to implement by-value loading with the ability to detect `NULL` values.

To illustrate this functionality, consider the following view that load the employee's residence country by value:

```
#pragma db view object(employee) \
  object(country = res: employee::residence_) transient
struct employee_res_country
{
  using country_ptr = country*;

  #pragma db member(res_) virtual(country_ptr) get(&this.res) \
    set(this.res_null = ((?) == nullptr))

  country res;
  bool res_null;
};
```

Here we are using a virtual data member (Section 14.4.13, "`virtual`") to add an object pointer member to the view. Its accessor expression returns the pointer to the `res` member so that the implementation can load the data into it. The modifier expression checks the passed pointer to initialize the `NULL` value indicator. Here, the two possible values that can be passed to the modifier expression are the address of the `res` member that we returned earlier from the accessor and `NULL` (strictly speaking, there is a third possibility: the address of an object that was found in the session cache).

If we are not interested in the NULL indicator, then the above view can simplified to this:

```
#pragma db view object(employee) \
  object(country = res: employee::residence_) transient
struct employee_res_country
{
  using country_ptr = country*;

  #pragma db member(res_) virtual(country_ptr) get(&this.res) set()

  country res;
};
```

That is, we specify an empty modifier expression which leads to the value being ignored.

As another example of by-value loading, consider a view that allows us to load objects into existing instances that have been allocated outside the view:

```
#pragma db view object(employee)                \
  object(country = res: employee::residence_)   \
  object(country = nat: employee::nationality_)
struct employee_country
{
  employee_country (country& r, country& n): res (&r), nat (&n) {}

  country* res;
  country* nat;
};
```

And here is how we can use this view:

```
using result = odb::result<employee_country>;

transaction t (db.begin ());

result r (db.query<employee_country> (...);

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  country res, nat;
  employee_country v (res, nat);
  i.load (v);

  if (v.res != nullptr)
    ... // Result is in res.

  if (v.nat != nullptr)
```

```
    ... // Result is in nat.
}

t.commit ();
```

As a final example of the by-value loading, consider the following view which implements a slightly more advanced logic: if the object is already in the session cache, then it sets the pointer data member in the view (`er_p`) to that. Otherwise, it loads the data into the by-value instance (`er`). We can also check whether the pointer data member points to the instance to distinguish between the two outcomes. And we can check it for `nullptr` to detect `NULL` values.

```
#pragma db view object(employer)
struct employer_view
{
  // Since we may be getting the pointer as both smart and raw, we
  // need to create a bit of support code to use in the modifier
  // expression.
  //
  void set_er (employer* p) {er_p = p;}                    // &er or NULL.
  void set_er (shared_ptr<employer> p) {er_p = p.get ();} // From cache.

  #pragma db get(&this.er) set(set_er(?))
  employer* er_p;

  #pragma db transient
  employer er;

  // Return-by-value support (e.g., query_value()).
  //
  employer_view (): er_p (0) {}
  employer_view (const employer_view& x)
    : er_p (x.er_p == &x.er ? &er : x.er_p), er (x.er) {}
};
```

We can use object loading views with polymorphic objects (Section 8.2, "Polymorphism Inheritance"). Note, however, that when loading a derived object via the base pointer in a view, a separate statement will be executed to load the dynamic part of the object. There is no support for by-value loading for polymorphic objects.

We can also use object loading views with objects without id (Section 14.1.6, "`no_id`"). Note, however, that for such objects, `NULL` values are not automatically detected (since there is no primary key, which is otherwise guaranteed to be not `NULL`, there might not be a column on which to base this detection). The workaround for this limitation is to load an otherwise not `NULL` column next to the object which will serve as an indicator. For example:

```
#pragma db object no_id
class object
{
  ...

  int n; // NOT NULL
  std::string s;
};

#include <odb/nullable.hxx>

#pragma db view object(object)
struct view
{

  odb::nullable<int> n; // If 'n' is NULL, then, logically, so is 'o'.
  unique_ptr<object> o;
};
```

10.3 Table Views

A table view is similar to an object view except that it is based on one or more database tables instead of persistent objects. Table views are primarily useful when dealing with ad-hoc tables that are not mapped to persistent classes.

To associate one or more tables with a view we use the `db table` pragma (Section 14.2.2, "`table`"). To associate the second and subsequent tables we repeat the `db table` pragma for each additional table. For example, the following view is based on the `employee_extra` legacy table we have defined at the beginning of the chapter.

```
#pragma db view table("employee_extra")
struct employee_vacation
{
  #pragma db column("employee_id") type("INTEGER")
  unsigned long long employee_id;

  #pragma db column("vacation_days") type("INTEGER")
  unsigned short vacation_days;
};
```

Besides the table name in the `db table` pragma we also have to specify the column name for each view data member. Note that unlike for object views, the ODB compiler does not try to automatically come up with column names for table views. Furthermore, we cannot use references to object members either, since there are no associated objects in table views. Instead, the actual column name or column expression must be specified as a string literal. The column name can also be qualified with a table name either in the `"table.column"` form or, if either a table or a column name contains a period, in the `"table"."column"` form. The following example

illustrates the use of a column expression:

```
#pragma db view table("employee_extra")
struct employee_max_vacation
{
  #pragma db column("max(vacation_days)") type("INTEGER")
  unsigned short max_vacation_days;
};
```

Both the associated table names and the column names can be qualified with a database schema, for example:

```
#pragma db view table("hr.employee_extra")
struct employee_max_vacation
{
  #pragma db column("hr.employee_extra.vacation_days") type("INTEGER")
  unsigned short vacation_days;
};
```

For more information on database schemas and the format of the qualified names, refer to Section 14.1.8, "`schema`".

Note also that in the above examples we specified the SQL type for each of the columns to make sure that the ODB compiler has knowledge of the actual types as specified in the database schema. This is required to obtain correct and optimal generated code.

The complete syntax of the `db table` pragma is similar to the `db object` pragma and is shown below:

`table("`*`name`***`" [= "`***`alias`***`"] [`***`join-type`***`] [: `***`join-condition`***`])`**

The *name* part is a database table name. The optional *alias* part gives this table an alias. If provided, the alias must be used instead of the table whenever a reference to a table is used. Contexts where such a reference may be needed include the join condition (discussed below), column names, and query expressions. The optional *join-type* part specifies the way this table is associated. It can be `left`, `right`, `full`, `inner`, and `cross` with `left` being the default. Finally, the optional *join-condition* part provides the criteria which should be used to associate this table with any of the previously associated tables or, as we will see in Section 10.4, "Mixed Views", objects. Note that while the first associated table can have an alias, it cannot have a join type or condition.

Similar to object views, for each subsequent associated table the ODB compiler needs a join condition. However, unlike for object views, for table views the ODB compiler does not try to come up with one automatically. Furthermore, we cannot use references to object members corresponding to object relationships either, since there are no associated objects in table views. Instead, for each subsequent associated table, a join condition must be specified as a custom

query expression. While the syntax of the query expression is the same as in the query facility used to query the database for objects (Chapter 4, "Querying the Database"), a join condition for a table is normally specified as a single string literal containing a native SQL query expression.

As an example of a multi-table view, consider the `employee_health` table that we define in addition to `employee_extra`:

```
CREATE TABLE employee_health(
  employee_id INTEGER NOT NULL,
  sick_leave_days INTEGER NOT NULL)
```

Given these two tables we can now define a view that returns both the vacation and sick leave information for each employee:

```
#pragma db view table("employee_extra" = "extra") \
  table("employee_health" = "health": \
        "extra.employee_id = health.employee_id")
struct employee_leave
{
  #pragma db column("extra.employee_id") type("INTEGER")
  unsigned long long employee_id;

  #pragma db column("vacation_days") type("INTEGER")
  unsigned short vacation_days;

  #pragma db column("sick_leave_days") type("INTEGER")
  unsigned short sick_leave_days;
};
```

Querying the database for a table view is the same as for an object view except that we can only use native query expressions. For example:

```
using query = odb::query<employee_leave>;
using result = odb::result<employee_leave>;

transaction t (db.begin ());

unsigned short v_min = ...
unsigned short l_min = ...

result r (db.query<employee_leave> (
  "vacation_days > " + query::_val(v_min) + "AND" +
  "sick_leave_days > " + query::_val(l_min)));

t.commit ();
```

10.4 Mixed Views

A mixed view has both associated objects and tables. As a first example of a mixed view, let us improve `employee_vacation` from the previous section to return the employee's first and last names instead of the employee id. To achieve this we have to associate both the `employee` object and the `employee_extra` table with the view:

```
#pragma db view object(employee) \
  table("employee_extra" = "extra": "extra.employee_id = " + employee::id_)
struct employee_vacation
{
  std::string first;
  std::string last;

  #pragma db column("extra.vacation_days") type("INTEGER")
  unsigned short vacation_days;
};
```

When querying the database for a mixed view, we can use query members for the parts of the query expression that involves object members but have to fall back to using the native syntax for the parts that involve table columns. For example:

```
using query = odb::query<employee_vacation>;
using result = odb::result<employee_vacation>;

transaction t (db.begin ());

result r (db.query<employee_vacation> (
  (query::last == "Doe") + "AND extra.vacation_days <> 0"));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cout << i->first << " " << i->last << " " << i->vacation_days << endl;

t.commit ();
```

As another example, consider a more advanced view that associates two objects via a legacy table. This view allows us to find the previous employer name for each employee:

```
#pragma db view object(employee) \
  table("employee_extra" = "extra": "extra.employee_id = " + employee::id_) \
  object(employer: "extra.previous_employer_id = " + employer::id_)
struct employee_prev_employer
{
  std::string first;
  std::string last;

  // If previous_employer_id is NULL, then the name will be NULL as well.
  // We use the odb::nullable wrapper to handle this.
```

```
  //
  #pragma db column(employer::name_)
  odb::nullable<std::string> prev_employer_name;
};
```

10.5 View Query Conditions

Object, table, and mixed views can also specify an optional query condition that should be used whenever the database is queried for this view. To specify a query condition we use the `db query` pragma (Section 14.2.3, "`query`").

As an example, consider a view that returns some information about all the employees that are over a predefined retirement age. One way to implement this would be to define a standard object view as we have done in the previous sections and then use a query like this:

```
result r (db.query<employee_retirement> (query::age > 50));
```

The problem with the above approach is that we have to keep repeating the `query::age > 50` expression every time we execute the query, even though this expression always stays the same. View query conditions allow us to solve this problem. For example:

```
#pragma db view object(employee) query(employee::age > 50)
struct employee_retirement
{
  std::string first;
  std::string last;
  unsigned short age;
};
```

With this improvement we can rewrite our query like this:

```
result r (db.query<employee_retirement> ());
```

But what if we may also need to restrict the result set based on some varying criteria, such as the employee's last name? Or, in other words, we may need to combine a constant query expression specified in the `db query` pragma with the varying expression specified at the query execution time. To allow this, the `db query` pragma syntax supports the use of the special `(?)` placeholder that indicates the position in the constant query expression where the runtime expression should be inserted. For example:

```
#pragma db view object(employee) query(employee::age > 50 && (?))
struct employee_retirement
{
  std::string first;
  std::string last;
  unsigned short name;
};
```

With this change we can now use additional query criteria in our view:

```
result r (db.query<employee_retirement> (query::last == "Doe"));
```

The syntax of the expression in a query condition is the same as in the query facility used to query the database for objects (Chapter 4, "Querying the Database") except for two differences. Firstly, for query members, instead of using `odb::query<object>::member` names, we refer directly to object members, using the object alias instead of the object name if an alias was assigned. Secondly, query conditions support the special `(?)` placeholder which can be used both in the C++-integrated query expressions as was shown above and in native SQL expressions specified as string literals. The following view is an example of the latter case:

```
#pragma db view table("employee_extra") \
  query("vacation_days <> 0 AND (?)")
struct employee_vacation
{
  ...
};
```

Another common use case for query conditions are views with the `ORDER BY` or `GROUP BY` clause. Such clauses are normally present in the same form in every query involving such views. As an example, consider an aggregate view which calculate the minimum and maximum ages of employees for each employer:

```
#pragma db view object(employee) object(employer) \
  query((?) + "GROUP BY" + employer::name_)
struct employer_age
{
  #pragma db column(employer::name_)
  std::string employer_name;

  #pragma db column("min(" + employee::age_ + ")")
  unsigned short min_age;

  #pragma db column("max(" + employee::age_ + ")")
  unsigned short max_age;
};
```

The query condition can be optionally followed (or replaced, if no constant query expression is needed) by one or more *result modifiers*. Currently supported result modifiers are `distinct` (which is translated to `SELECT DISTINCT`) and `for_update` (which is translated to `FOR UPDATE` or equivalent for database systems that support it). As an example, consider a view that allows us to get some information about employers ordered by the object id and without any duplicates:

```
#pragma db view object(employer) object(employee) \
  query((?) + "ORDER BY" + employer::name_, distinct)
struct employer_info
{
  ...
};
```

If we don't require ordering, then this view can be re-implemented like this:

```
#pragma db view object(employer) object(employee) query(distinct)
struct employer_info
{
  ...
};
```

10.6 Native Views

The last kind of view supported by ODB is a native view. Native views are a low-level mechanism for capturing results of native SQL queries, stored procedure calls, etc. Native views don't have associated tables or objects. Instead, we use the `db query` pragma to specify the native SQL query, which should normally include the select-list and, if applicable, the from-list. For example, here is how we can re-implement the `employee_vacation` table view from Section 10.3 above as a native view:

```
#pragma db view query("SELECT employee_id, vacation_days " \
                      "FROM employee_extra")
struct employee_vacation
{
  #pragma db type("INTEGER")
  unsigned long long employee_id;

  #pragma db type("INTEGER")
  unsigned short vacation_days;
};
```

In native views the columns in the query select-list are associated with the view data members in the order specified. That is, the first column is stored in the first member, the second column — in the second member, and so on. The ODB compiler does not perform any error checking in this association. As a result you must make sure that the number and order of columns in the query select-list match the number and order of data members in the view. This is also the reason why we are not required to provide the column name for each data member in native views, as is the case for object and table views.

Note also that while it is always possible to implement a table view as a native view, the table views must be preferred since they are safer. In a native view, if you add, remove, or rearrange data members without updating the column list in the query, or vice versa, at best, this will result in a runtime error. In contrast, in a table view such changes will result in the query being auto-

matically updated.

Similar to object and table views, the query specified for a native view can contain the special `(?)` placeholder which is replaced with the query expression specified at the query execution time. If the native query does not contain a placeholder, as in the example above, then any query expression specified at the query execution time is appended to the query text along with the `WHERE` keyword, if required. The following example shows the usage of the placeholder:

```
#pragma db view query("SELECT employee_id, vacation_days " \
                      "FROM employee_extra " \
                      "WHERE vacation_days <> 0 AND (?)")
struct employee_vacation
{
  ...
};
```

As another example, consider a view that returns the next value of a database sequence:

```
#pragma db view query("SELECT nextval('my_seq')")
struct sequence_value
{
  unsigned long long value;
};
```

While this implementation can be acceptable in some cases, it has a number of drawbacks. Firstly, the name of the sequence is fixed in the view, which means if we have a second sequence, we will have to define another, almost identical view. Similarly, the operation that we perform on the sequence is also fixed. In some situations, instead of returning the next value, we may need the last value.

Note that we cannot use the placeholder mechanism to resolve these problems since placeholders can only be used in the `WHERE`, `GROUP BY`, and similar clauses. In other words, the following won't work:

```
#pragma db view query("SELECT nextval('(?)')")
struct sequence_value
{
  unsigned long long value;
};

result r (db.query<sequence_value> ("my_seq"));
```

To support these kinds of use cases, ODB allows us to specify the complete query for a native view at runtime rather than at the view definition. To indicate that a native view has a runtime query, we can either specify the empty `db query` pragma or omit the pragma altogether. For example:

```
#pragma db view
struct sequence_value
{
  unsigned long long value;
};
```

Given this view, we can perform the following queries:

```
using query = odb::query<sequence_value>;
using result = odb::result<sequence_value>;

string seq_name = ...

result l (db.query<sequence_value> (
  "SELECT lastval('" + seq_name + "')"));

result n (db.query<sequence_value> (
  "SELECT nextval('" + seq_name + "')"));
```

Native views can also be used to call and handle results of stored procedures. The semantics and limitations of stored procedures vary greatly between database systems while some do not support this functionality at all. As a result, support for calling stored procedures using native views is described for each database system in Part II, "Database Systems".

10.7 Other View Features and Limitations

Views cannot be derived from other views. However, you can derive a view from a transient C++ class. View data members cannot be object pointers. If you need to access data from a pointed-to object, then you will need to associate such an object with the view. Similarly, view data members cannot be containers. These two limitations also apply to composite value types that contain object pointers or containers. Such composite values cannot be used as view data members.

On the other hand, composite values that do not contain object pointers or containers can be used in views. As an example, consider a modified version of the `employee` persistent class that stores a person's name as a composite value:

```
#pragma db value
class person_name
{
  std::string first_;
  std::string last_;
};

#pragma db object
class employee
{
  ...
```

```
  person_name name_;

  ...
};
```

Given this change, we can re-implement the `employee_name` view like this:

```
#pragma db view object(employee)
struct employee_name
{
  person_name name;
};
```

It is also possible to extract some or all of the nested members of a composite value into individual view data members. Here is how we could have defined the `employee_name` view if we wanted to keep its original structure:

```
#pragma db view object(employee)
struct employee_name
{
  #pragma db column(employee::name.first_)
  std::string first;

  #pragma db column(employee::name.last_)
  std::string last;
};
```

11 Session

A session is an application's unit of work that may encompass several database transactions. In this version of ODB a session is just an object cache. In future versions it may provide additional functionality, such as delayed database operations and automatic object state change tracking. As discussed later in Section 11.2, "Custom Sessions", it is also possible to provide a custom session implementation that provides these or other features.

Session support is optional and can be enabled or disabled on the per object basis using the `db session` pragma, for example:

```
#pragma db object session
class person
{
  ...
};
```

We can also enable or disable session support for a group of objects at the namespace level:

```
#pragma db namespace session
namespace accounting
{
  #pragma db object                // Session support is enabled.
  class employee
  {
    ...
  };

  #pragma db object session(false) // Session support is disabled.
  class employer
  {
    ...
  };
}
```

Finally, we can pass the `--generate-session` ODB compiler option to enable session support by default. With this option session support will be enabled for all the persistent classes except those for which it was explicitly disabled using the `db session`. An alternative to this method with the same effect is to enable session support for the global namespace:

```
#pragma db namespace() session
```

Each thread of execution in an application can have only one active session at a time. A session is started by creating an instance of the `odb::session` class and is automatically terminated when this instance is destroyed. You will need to include the `<odb/session.hxx>` header file to make this class available in your application. For example:

```
#include <odb/database.hxx>
#include <odb/session.hxx>
#include <odb/transaction.hxx>

using namespace odb::core;

{
  session s;

  // First transaction.
  //
  {
    transaction t (db.begin ());
    ...
    t.commit ();
  }

  // Second transaction.
  //
  {
    transaction t (db.begin ());
    ...
    t.commit ();
  }

  // Session 's' is terminated here.
}
```

The `session` class has the following interface:

```
namespace odb
{
  class session
  {
  public:
    session (bool make_current = true);
    ~session ();

    // Copying or assignment of sessions is not supported.
    //
  private:
    session (const session&);
    session& operator= (const session&);

    // Current session interface.
    //
  public:
    static session&
    current ();

    static bool
```

```
    has_current ();

    static void
    current (session&);

    static void
    reset_current ();

    static session*
    current_pointer ();

    static void
    current_pointer (session*);

    // Object cache interface.
    //
  public:
    template <typename T>
    struct cache_position {...};

    template <typename T>
    cache_position<T>
    cache_insert (database&,
                  const object_traits<T>::id_type&,
                  const object_traits<T>::pointer_type&);

    template <typename T>
    object_traits<T>::pointer_type
    cache_find (database&, const object_traits<T>::id_type&) const;

    template <typename T>
    void
    cache_erase (const cache_position<T>&);

    template <typename T>
    void
    cache_erase (database&, const object_traits<T>::id_type&);
  };
}
```

The session constructor creates a new session and, if the `make_current` argument is `true`, sets it as a current session for this thread. If we try to make a session current while there is already another session in effect for this thread, then the constructor throws the `odb::already_in_session` exception. The destructor clears the current session for this thread if this session is the current one.

The static `current()` accessor returns the currently active session for this thread. If there is no active session, this function throws the `odb::not_in_session` exception. We can check whether there is a session in effect in this thread using the `has_current()` static function.

The static `current()` modifier allows us to set the current session for this thread. The `reset_current()` static function clears the current session. These two functions allow for more advanced use cases, such as multiplexing two or more sessions on the same thread.

The static `current_pointer()` overloaded functions provided the same functionality but using pointers. Specifically, the `current_pointer()` accessor can be used to test whether there is a current session and get a pointer to it all with a single call.

We normally don't use the object cache interface directly. However, it could be useful in some cases, for example, to find out whether an object has already been loaded. Note that when calling `cache_insert()`, `cache_find()`, or the second version of `cache_erase()`, you need to specify the template argument (object type) explicitly. It is also possible to access the underlying cache data structures directly. This can be useful if, for example, you want to iterate over the objects store in the cache. Refer to the ODB runtime header files for more details on this direct access.

11.1 Object Cache

A session is an object cache. Every time a session-enabled object is made persistent by calling the `database::persist()` function (Section 3.8, "Making Objects Persistent"), loaded by calling the `database::load()` or `database::find()` function (the pointer-returning overloads only; Section 3.9, "Loading Persistent Objects"), or loaded by iterating over a query result (Section 4.4, "Query Result"), the pointer to the persistent object, in the form of the canonical object pointer (Section 3.3, "Object and View Pointers"), is stored in the session. For as long as the session is in effect, any subsequent calls to load the same object will return the cached instance. When an object's state is deleted from the database with the `database::erase()` function (Section 3.11, "Deleting Persistent Objects"), the cached object pointer is removed from the session. For example:

```
shared_ptr<person> p (new person ("John", "Doe"));

session s;
transaction t (db.begin ());

unsigned long long id (db.persist (p));             // p is cached in s.
shared_ptr<person> p1 (db.load<person> (id)); // p1 same as p.

t.commit ();
```

The per-object caching policies depend on the object pointer kind (Section 6.5, "Using Custom Smart Pointers"). Objects with a unique pointer, such as `std::auto_ptr` or `std::unique_ptr`, as an object pointer are never cached since it is not possible to have two such pointers pointing to the same object. When an object is persisted via a pointer or loaded as a dynamically allocated instance, objects with both raw and shared pointers as object pointers are cached. If an object is persisted as a reference or loaded into a pre-allocated instance, the object is

only cached if its object pointer is a raw pointer.

Also note that when we persist an object as a constant reference or constant pointer, the session caches such an object as unrestricted (non-`const`). This can lead to undefined behavior if the object being persisted was actually created as `const` and is later found in the session cache and used as non-`const`. As a result, when using sessions, it is recommended that all persistent objects be created as non-`const` instances. The following code fragment illustrates this point:

```
void save (database& db, shared_ptr<const person> p)
{
  transaction t (db.begin ());
  db.persist (p); // Persisted as const pointer.
  t.commit ();
}

session s;

shared_ptr<const person> p1 (new const person ("John", "Doe"));
unsigned long long id1 (save (db, p1)); // p1 is cached in s as non-const.

{
  transaction t (db.begin ());
  shared_ptr<person> p (db.load<person> (id1)); // p == p1
  p->age (30); // Undefined behavior since p1 was created const.
  t.commit ();
}

shared_ptr<const person> p2 (new person ("Jane", "Doe"));
unsigned long long id2 (save (db, p2)); // p2 is cached in s as non-const.

{
  transaction t (db.begin ());
  shared_ptr<person> p (db.load<person> (id2)); // p == p2
  p->age (30); // Ok, since p2 was not created const.
  t.commit ();
}
```

11.2 Custom Sessions

ODB can use a custom session implementation instead of the default `odb::session`. There could be multiple reasons for an application to provide its own session. For example, the application may already include a notion of an object cache or registry which ODB can re-use. A custom session can also provide additional functionality, such as automatic change tracking, delayed database operations, or object eviction. Finally, the session-per-thread approach used by `odb::session` may not be suitable for all applications. For instance, some may need a thread-safe session that can be shared among multiple threads. For an example of a custom session that implements automatic change tracking by keeping original copies of the objects, refer

to the `common/session/custom` test in the `odb-tests` package.

To use a custom session we need to specify its type with the `--session-type` ODB compiler command line option. We also need to include its definition into the generated header file. This can be achieved with the `--hxx-prologue` option. For example, if our custom session is called `app::session` and is defined in the `app/session.hxx` header file, then the corresponding ODB compiler options would look like this:

```
odb --hxx-prologue "#include \"app/session.hxx\"" \
--session-type ::app::session ...
```

A custom session should provide the following interface:

```
class custom_session
{
public:
  static bool
  _has_cache ();

  // Cache management functions.
  //
  template <typename T>
  struct cache_position
  {
    ...
  };

  template <typename T>
  static cache_position<T>
  _cache_insert (odb::database&,
                 const typename odb::object_traits<T>::id_type&,
                 const typename odb::object_traits<T>::pointer_type&);

  template <typename T>
  static typename odb::object_traits<T>::pointer_type
  _cache_find (odb::database&,
               const typename odb::object_traits<T>::id_type&);

  template <typename T>
  static void
  _cache_erase (const cache_position<T>&);

  // Notification functions.
  //
  template <typename T>
  static void
  _cache_persist (const cache_position<T>&);

  template <typename T>
  static void
```

```
  _cache_load (const cache_position<T>&);

  template <typename T>
  static void
  _cache_update (odb::database&, const T& obj);

  template <typename T>
  static void
  _cache_erase (odb::database&,
                const typename odb::object_traits<T>::id_type&);
};
```

The `_has_cache()` function shall return `true` if the object cache is in effect in the current thread.

The `cache_position` class template represents a position in the cache of the inserted object. It should be default and copy-constructible as well as copy-assignable. The default constructor shall create a special empty/`NULL` position. A call of any of the cache management or notification functions with such an empty/`NULL` position shall be ignored.

The `_cache_insert()` function shall add the object into the object cache and return its position. The `_cache_find()` function looks an object up in the object cache given its id. It returns a `NULL` pointer if the object is not found. The `_cache_erase()` cache management function shall remove the object from the cache. It is called if the database operation that caused the object to be inserted (for example, load) failed. Note also that after insertion the object state is undefined. You can only access the object state (for example, make a copy or clear a flag) from one of the notification functions discussed below.

The notification functions are called after an object has been persisted, loaded, updated, or erased, respectively. If your session implementation does not need some of the notifications, you still have to provide their functions, however, you can leave their implementations empty.

Notice also that all the cache management and notification functions are static. This is done in order to allow for a custom notion of a current session. Normally, the first step a non-empty implementation will perform is lookup the current session.

12 Optimistic Concurrency

The ODB transaction model (Section 3.5, "Transactions") guarantees consistency as long as we perform all the database operations corresponding to a specific application transaction in a single database transaction. That is, if we load an object within a database transaction and update it in the same transaction, then we are guaranteed that the object state that we are updating in the database is exactly the same as the state we have loaded. In other words, it is impossible for another process or thread to modify the object state in the database between these load and update operations.

In this chapter we use the term *application transaction* to refer to a set of operations on persistent objects that an application needs to perform in order to implement some application-specific functionality. The term *database transaction* refers to the set of database operations performed between the ODB `begin()` and `commit()` calls. Up until now we have treated application transactions and database transactions as essentially the same thing.

While this model is easy to understand and straightforward to use, it may not be suitable for applications that have long application transactions. The canonical example of such a situation is an application transaction that requires user input between loading an object and updating it. Such an operation may take an arbitrary long time to complete and performing it within a single database transaction will consume database resources as well as prevent other processes/threads from updating the object for too long.

The solution to this problem is to break up the long-lived application transaction into several short-lived database transactions. In our example that would mean loading the object in one database transaction, waiting for user input, and then updating the object in another database transaction. For example:

```
unsigned long long id = ...;
person p;

{
  transaction t (db.begin ());
  db.load (id, p);
  t.commit ();
}

cerr << "enter age for " << p.first () << " " << p.last () << endl;
unsigned short age;
cin >> age;
p.age (age);

{
```

```
  transaction t (db.begin ());
  db.update (p);
  t.commit ();
}
```

This approach works well if we only have one process/thread that can ever update the object. However, if we have multiple processes/threads modifying the same object, then this approach does not guarantee consistency anymore. Consider what happens in the above example if another process updates the person's last name while we are waiting for the user input. Since we loaded the object before this change occured, our version of the person's data will still have the old name. Once we receive the input from the user, we go ahead and update the object, overwriting both the old age with the new one (correct) and the new name with the old one (incorrect).

While there is no way to restore the consistency guarantee in an application transaction that consists of multiple database transactions, ODB provides a mechanism, called optimistic concurrency, that allows applications to detect and potentially recover from such inconsistencies.

In essence, the optimistic concurrency model detects mismatches between the current object state in the database and the state when it was loaded into the application memory. Such a mismatch would mean that the object was changed by another process or thread. There are several ways to implement such state mismatch detection. Currently, ODB uses object versioning while other methods, such as timestamps, may be supported in the future.

To declare a persistent class with the optimistic concurrency model we use the `optimistic` pragma (Section 14.1.5, "`optimistic`"). We also use the `version` pragma (Section 14.4.16, "`version`") to specify which data member will store the object version. For example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version
  unsigned long long version_;
};
```

The version data member is managed by ODB. It is initialized to `1` when the object is made persistent and incremented by `1` with each update. The `0` version value is not used by ODB and the application can use it as a special value, for example, to indicate that the object is transient. Note that for optimistic concurrency to function properly, the application should not modify the version member after making the object persistent or loading it from the database and until deleting the state of this object from the database. To avoid any accidental modifications to the version member, we can declare it `const`, for example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version
  const unsigned long long version_;
};
```

When we call the `database::update()` function (Section 3.10, "Updating Persistent Objects") and pass an object that has an outdated state, the `odb::object_changed` exception is thrown. At this point the application has two recovery options: it can abort and potentially restart the application transaction or it can reload the new object state from the database, re-apply or merge the changes, and call `update()` again. Note that aborting an application transaction that performs updates in multiple database transactions may require reverting changes that have already been committed to the database. As a result, this strategy works best if all the updates are performed in the last database transaction of the application transaction. This way the changes can be reverted by simply rolling back this last database transaction.

The following example shows how we can reimplement the above transaction using the second recovery option:

```
unsigned long long id = ...;
person p;

{
  transaction t (db.begin ());
  db.load (id, p);
  t.commit ();
}

cerr << "enter age for " << p.first () << " " << p.last () << endl;
unsigned short age;
cin >> age;
p.age (age);

{
  transaction t (db.begin ());

  try
  {
    db.update (p);
  }
  catch (const object_changed&)
  {
    db.reload (p);
    p.age (age);
    db.update (p);
```

```
  }

  t.commit ();
}
```

An important point to note in the above code fragment is that the second `update()` call cannot throw the `object_changed` exception because we are reloading the state of the object and updating it within the same database transaction.

Depending on the recovery strategy employed by the application, an application transaction with a failed update can be significantly more expensive than a successful one. As a result, optimistic concurrency works best for situations with low to medium contention levels where the majority of the application transactions complete without update conflicts. This is also the reason why this concurrency model is called optimistic.

In addition to updates, ODB also performs state mismatch detection when we are deleting an object from the database (Section 3.11, "Deleting Persistent Objects"). To understand why this can be important, consider the following application transaction:

```
unsigned long long id = ...;
person p;

{
  transaction t (db.begin ());
  db.load (id, p);
  t.commit ();
}

string answer;
cerr << "age is " << p.age () << ", delete?" << endl;
getline (cin, answer);

if (answer == "yes")
{
  transaction t (db.begin ());
  db.erase (p);
  t.commit ();
}
```

Consider again what happens if another process or thread updates the object by changing the person's age while we are waiting for the user input. In this case, the user makes the decision based on a certain age while we may delete (or not delete) an object that has a completely different age. Here is how we can fix this problem using optimistic concurrency:

```
unsigned long long id = ...;
person p;

{
```

```
  transaction t (db.begin ());
  db.load (id, p);
  t.commit ();
}

string answer;
for (bool done (false); !done; )
{
  if (answer.empty ())
    cerr << "age is " << p.age () << ", delete?" << endl;
  else
    cerr << "age changed to " << p.age () << ", still delete?" << endl;

  getline (cin, answer);

  if (answer == "yes")
  {
    transaction t (db.begin ());

    try
    {
      db.erase (p);
      done = true;
    }
    catch (const object_changed&)
    {
      db.reload (p);
    }

    t.commit ();
  }
  else
    done = true;
}
```

Note that state mismatch detection is performed only if we delete an object by passing the object instance to the `erase()` function. If we want to delete an object with the optimistic concurrency model regardless of its state, then we need to use the `erase()` function that deletes an object given its id, for example:

```
{
  transaction t (db.begin ());
  db.erase (p.id ());
  t.commit ();
}
```

Finally, note that for persistent classes with the optimistic concurrency model both the `update()` function as well as the `erase()` function that accepts an object instance as its argument no longer throw the `object_not_persistent` exception if there is no such object in the database. Instead, this condition is treated as a change of object state and the

`object_changed` exception is thrown instead.

For complete sample code that shows how to use optimistic concurrency, refer to the `optimistic` example in the `odb-examples` package.

13 Database Schema Evolution

When we add new persistent classes or change the existing ones, for example, by adding or deleting data members, the database schema necessary to store the new object model changes as well. At the same time, we may have existing databases that contain existing data. If new versions of your application don't need to handle old databases, then the schema creating functionality is all that you need. However, most applications will need to work with data stored by older versions of the same application.

We will call *database schema evolution* the overall task of updating the database to match the changes in the object model. Schema evolution usually consists of two sub-tasks: *schema migration* and *data migration*. Schema migration modifies the database schema to correspond to the current object model. In a relational database, this, for example, could require adding or dropping tables and columns. The data migration task involves converting the data stored in the existing database from the old format to the new one.

If performed manually, database schema evolution is a tedious and error-prone task. As a result, ODB provides comprehensive support for automated or, more precisely, semi-automated schema evolution. Specifically, ODB does fully-automatic schema migration and provides facilities to help you with data migration.

The topic of schema evolution is a complex and sensitive issue since normally there would be valuable, production data at stake. As a result, the approach taken by ODB is to provide simple and bullet-proof elementary building blocks (or migration steps) that we can understand and trust. Using these elementary blocks we can then implement more complex migration scenarios. In particular, ODB does not try to handle data migration automatically since in most cases this requires understanding of application-specific semantics. In other words, there is no magic.

There are two general approaches to working with older data: the application can either convert it to correspond to the new format or it can be made capable of working with multiple versions of this format. There is also a hybrid approach where the application may convert the data to the new format gradually as part of its normal functionality. ODB is capable of handling all these scenarios. That is, there is support for working with older models without performing any migration (schema or data). Alternatively, we can migrate the schema after which we have the choice of either also immediately migrating the data (*immediate data migration*) or doing it gradually (*gradual data migration*).

Schema evolution is already a complex task and we should not unnecessarily use a more complex approach where a simpler one would be sufficient. From the above, the simplest approach is the immediate schema migration that does not require any data migration. An example of such a change would be adding a new data member with the default value (Section 14.3.4, "`default`"). This case ODB can handle completely automatically.

If we do require data migration, then the next simplest approach is the immediate schema and data migration. Here we have to write custom migration code. However, it is separate from the rest of the core application logic and is executed at a well defined point (database migration). In other words, the core application logic need not be aware of older model versions. The potential drawback of this approach is performance. It may take a lot of resources and/or time to convert all the data upfront.

If the immediate migration is not possible, then the next option is the immediate schema migration followed by the gradual data migration. With this approach, both old and new data must co-exist in the new database. We also have to change the application logic to both account for different sources of the same data (for example, when either an old or new version of the object is loaded) as well as migrate the data when appropriate (for example, when the old version of the object is updated). At some point, usually when the majority of the data has been converted, gradual migrations are terminated with an immediate migration.

The most complex approach is working with multiple versions of the database without performing any migrations, schema or data. ODB does provide support for implementing this approach (Section 13.4, "Soft Object Model Changes"), however we will not cover it any further in this chapter. Generally, this will require embedding knowledge about each version into the core application logic which makes it hard to maintain for any non-trivial object model.

Note also that when it comes to data migration, we can use the immediate variant for some changes and gradual for others. We will discuss various migration scenarios in greater detail in section Section 13.3, "Data Migration".

13.1 Object Model Version and Changelog

To enable schema evolution support in ODB we need to specify the object model version, or, more precisely, two versions. The first is the base model version. It is the lowest version from which we will be able to migrate. The second version is the current model version. In ODB we can migrate from multiple previous versions by successively migrating from one to the next until we reach the current version. We use the `db model version` pragma to specify both the base and current versions.

When we enable schema evolution for the first time, our base and current versions will be the same, for example:

```
#pragma db model version(1, 1)
```

Once we release our application, its users may create databases with the schema corresponding to this version of the object model. This means that if we make any modifications to our object model that also change the schema, then we will need to be able to migrate the old databases to this new schema. As a result, before making any new changes after a release, we increment the current version, for example:

```
#pragma db model version(1, 2)
```

To put this another way, we can stay on the same version during development and keep adding new changes to it. But once we release it, any new changes to the object model will have to be done in a new version.

It is easy to forget to increment the version before making new changes to the object model. To help solve this problem, the `db model version` pragma accepts a third optional argument that specify whether the current version is open or closed for changes. For example:

```
#pragma db model version(1, 2, open)   // Can add new changes to
                                       // version 2.

#pragma db model version(1, 2, closed) // Can no longer add new
                                       // changes to version 2.
```

If the current version is closed, ODB will refuse to accept any new schema changes. In this situation you would normally increment the current version and mark it as open or you could re-open the existing version if, for example, you need to fix something. Note, however, that re-opening versions that have been released will most likely result in migration malfunctions. By default the version is open.

Normally, an application will have a range of older database versions from which it is able to migrate. When we change this range by removing support for older versions, we also need to adjust the base model version. This will make sure that ODB does not keep unnecessary information around.

A model version (both base and current) is a 64-bit unsigned integer (`unsigned long long`). `0` is reserved to signify special situations, such as the lack of schema in the database. Other than that, we can use any values as versions as long as they are monotonically increasing. In particular, we don't have to start with version `1` and can increase the versions by any increment.

One versioning approach is to use an independent object model version by starting from version `1` and also incrementing by `1`. The alternative is to make the model version correspond to the application version. For example, if our application is using the `X.Y.Z` version format, then we could encode it as a hexadecimal number and use that as our model version, for example:

```
#pragma db model version(0x020000, 0x020306) // 2.0.0-2.3.6
```

Most real-world object models will be spread over multiple header files and it will be burdensome to repeat the `db model version` pragma in each of them. The recommended way to handle this situation is to place the `version` pragma into a separate header file and include it into the object model files. If your project already has a header file that defines the application version, then it is natural to place this pragma there. For example:

```
// version.hxx
//
// Define the application version.
//

#define MYAPP_VERSION 0x020306 // 2.3.6

#ifdef ODB_COMPILER
#pragma db model version(1, 7)
#endif
```

Note that we can also use macros in the `version` pragma which allows us to specify all the versions in a single place. For example:

```
#define MYAPP_VERSION      0x020306 // 2.3.6
#define MYAPP_BASE_VERSION 0x020000 // 2.0.0

#ifdef ODB_COMPILER
#pragma db model version(MYAPP_BASE_VERSION, MYAPP_VERSION)
#endif
```

It is also possible to have multiple object models within the same application that have different versions. Such models must be independent, that is, no headers from one model shall include a header from another. You will also need to assign different schema names to each model with the `--schema-name` ODB compiler option.

Once we specify the object model version, the ODB compiler starts tracking database schema changes in a changelog file. Changelog has an XML-based, line-oriented format. It uses XML in order to provide human readability while also facilitating, if desired, processing and analysis with custom tools. The line orientation makes it easy to review with tools like `diff`.

The changelog is maintained by the ODB compiler. Specifically, you do not need to make any manual changes to this file. You will, however, need to keep it around from one invocation of the ODB compiler to the next. In other words, the changelog file is both the input and the output of the ODB compiler. This, for example, means that if your project's source code is stored in a version control repository, then you will most likely want to store the changelog there as well. If you delete the changelog, then any ability to do schema migration will be lost.

The only operation that you may want to perform with the changelog is to review the database schema changes that resulted from the C++ object model changes. For this you can use a tool like `diff` or, better yet, the change review facilities offered by your revision control system. For this purpose the contents of a changelog will be self-explanatory.

As an example, consider the following initial object model:

```
// person.hxx
//

#include <string>

#pragma db model version(1, 1)

#pragma db object
class person
{
  ...

  #pragma db id auto
  unsigned long long id_;

  std::string first_;
  std::string last_;
};
```

We then compile this header file with the ODB compiler (using the PostgreSQL database as an example):

```
odb --database pgsql --generate-schema person.hxx
```

If we now look at the list of generated files, then in addition to the now familiar `person-odb.?xx` and `person.sql`, we will also see `person.xml` — the changelog file. Just for illustration, below are the contents of this changelog.

```
<changelog database="pgsql">
  <model version="1">
    <table name="person" kind="object">
      <column name="id" type="BIGINT" null="false"/>
      <column name="first" type="TEXT" null="false"/>
      <column name="last" type="TEXT" null="false"/>
      <primary-key auto="true">
        <column name="id"/>
      </primary-key>
    </table>
  </model>
</changelog>
```

Let's say we now would like to add another data member to the `person` class — the middle name. We increment the version and make the change:

```
#pragma db model version(1, 2)

#pragma db object
class person
{
  ...
```

```
  #pragma db id auto
  unsigned long long id_;

  std::string first_;
  std::string middle_;
  std::string last_;
};
```

We use exactly the same command line to re-compile our file:

```
odb --database pgsql --generate-schema person.hxx
```

This time the ODB compiler will read the old changelog, update it, and write out the new version. Again, for illustration only, below are the updated changelog contents:

```
<changelog database="pgsql">
  <changeset version="2">
    <alter-table name="person">
      <add-column name="middle" type="TEXT" null="false"/>
    </alter-table>
  </changeset>

  <model version="1">
    <table name="person" kind="object">
      <column name="id" type="BIGINT" null="false"/>
      <column name="first" type="TEXT" null="false"/>
      <column name="last" type="TEXT" null="false"/>
      <primary-key auto="true">
        <column name="id"/>
      </primary-key>
    </table>
  </model>
</changelog>
```

Just to reiterate, while the changelog may look like it could be written by hand, it is maintained completely automatically by the ODB compiler and the only reason you may want to look at its contents is to review the database schema changes. For example, if we compare the above two changelogs with `diff`, we will get the following summary of the database schema changes:

```
--- person.xml.orig
+++ person.xml
@@ -1,4 +1,10 @@
<changelog database="pgsql">
+  <changeset version="2">
+    <alter-table name="person">
+      <add-column name="middle" type="TEXT" null="false"/>
+    </alter-table>
+  </changeset>
```

```
+
  <model version="1">
    <table name="person" kind="object">
      <column name="id" type="BIGINT" null="false"/>
```

The changelog is only written when we generate the database schema, that is, the `--generate-schema` option is specified. Invocations of the ODB compiler that only produce the database support code (C++) do not read or update the changelog. To put it another way, the changelog tracks changes in the resulting database schema, not the C++ object model.

ODB ignores column order when comparing database schemas. This means that we can re-order data members in a class without causing any schema changes. Member renames, however, will result in schema changes since the column name changes as well (unless we specified the column name explicitly). From ODB's perspective such a rename looks like the deletion of one data member and the addition of another. If we don't want this to be treated as a schema change, then we will need to keep the old column name by explicitly specifying it with the `db column` pragma. For example, here is how we can rename `middle_` to `middle_name_` without causing any schema changes:

```
#pragma db model version(1, 2)

#pragma db object
class person
{
  ...

  #pragma db column("middle") // Keep the original column name.
  std::string middle_name_;

  ...
};
```

If your object model consists of a large number of header files and you generate the database schema for each of them individually, then a changelog will be created for each of your header files. This may be what you want, however, the large number of changelogs can quickly become unwieldy. In fact, if you are generating the database schema as standalone SQL files, then you may have already experienced a similar problem caused by a large number of `.sql` files, one for each header.

The solution to both of these problems is to generate a combined database schema file and a single changelog. For example, assume we have three header files in our object model: `person.hxx`, `employee.hxx`, and `employer.hxx`. To generate the database support code we compile them as usual but without specifying the `--generate-schema` option. In this case no changelog is created or updated:

```
odb --database pgsql person.hxx
odb --database pgsql employee.hxx
odb --database pgsql employer.hxx
```

To generate the database schema, we perform a separate invocation of the ODB compiler. This time, however, we instruct it to only generate the schema (`--generate-schema-only`) and produce it combined (`--at-once`) for all the files in our object model:

```
odb --database pgsql --generate-schema-only --at-once \
--input-name company person.hxx employee.hxx employer.hxx
```

The result of the above command is a single `company.sql` file (the name is derived from the `--input-name` value) that contains the database schema for our entire object model. There is also a single corresponding changelog file — `company.xml`.

The same can be achieved for the embedded schema by instructing the ODB compiler to generate the database creation code into a separate C++ file (`--schema-format separate`):

```
odb --database pgsql --generate-schema-only --schema-format separate \
--at-once --input-name company person.hxx employee.hxx employer.hxx
```

The result of this command is a single `company-schema.cxx` file and, again, `company.xml`.

Note also that by default the changelog file is not placed into the directory specified with the `--output-dir` option. This is due to the changelog being both an input and an output file at the same time. As a result, by default, the ODB compiler will place it in the directory of the input header file.

There is, however, a number of command line options (including `--changelog-dir`) that allow us to fine-tune the name and location of the changelog file. For example, you can instruct the ODB compiler to read the changelog from one file while writing it to another. This, for example, can be useful if you want to review the changes before discarding the old file. For more information on these options, refer to the ODB Compiler Command Line Manual and search for "changelog".

When we were discussing version increments above, we used the terms *development* and *release*. Specifically, we talked about keeping the same object model versions during development periods and incrementing them after releases. What is a development period and a release in this context? These definitions can vary from project to project. Generally, during a development period we work on one or more changes to the object model that result in the changes to the database schema. A release is a point where we make our changes available to someone else who may have an older database to migrate from. In the traditional sense, a release is a point where you make a new version of your application available to its users. However, for schema evolution purposes, a release could also mean simply making your schema-altering changes available to

other developers on your team. Let us consider two common scenarios to illustrate how all this fits together.

One way to setup a project would be to re-use the application development period and application release for schema evolution. That is, during a new application version development we keep a single object model version and when we release the application, we increment the model version. In this case it makes sense to also reuse the application version as a model version for consistency. Here is a step-by-step guide for this setup:

1. During development, keep the current object model version open.
2. Before the release (for example, when entering a "feature freeze") close the version.
3. After the release, update the version and open it.
4. For each new feature, review the changeset at the top of the changelog, for example, with `diff` or your version control facilities. If you are using a version control, then this is best done just before committing your changes to the repository.

An alternative way to setup schema versioning in a project would be to define the development period as working on a single feature and the release as making this feature available to other people (developers, testers, etc.) on your team, for example, by committing the changes to a public version control repository. In this case, the object model version will be independent of the application version and can simply be a sequence that starts with `1` and is incremented by `1`. Here is a step-by-step guide for this setup:

1. Keep the current model version closed. Once a change is made that affects the database schema, the ODB compiler will refuse to update the changelog.
2. If the change is legitimate, open a new version, that is, increment the current version and make it open.
3. Once the feature is implemented and tested, review the final set of database changes (with `diff` or your version control facilities), close the version, and commit the changes to the version control repository (if using).

If you are using a version control repository that supports pre-commit checks, then you may want to consider adding such a check to make sure the committed version is always closed.

If we are just starting schema evolution in our project, which approach should we choose? The two approaches will work better in different situations since they have a different set of advantages and disadvantages. The first approach, which we can call version per application release, is best suited for simpler projects with smaller releases since otherwise a single migration will bundle a large number of unrelated actions corresponding to different features. This can become difficult to review and, if things go wrong, debug.

The second approach, which we can call version per feature, is much more modular and provides a number of additional benefits. We can perform migrations for each feature as a discreet step which makes it easier to debug. We can also place each such migration step into a separate transaction further improving reliability. It also scales much better in larger teams where multiple developers can work concurrently on features that affect the database schema. For example, if you find yourself in a situation where another developer on your team used the same version as you and managed to commit his changes before you (that is, you have a merge conflict), then you can simply change the version to the next available one, regenerate the changelog, and continue with your commit.

Overall, unless you have strong reasons to prefer the version per application release approach, rather choose version per feature even though it may seem more complex at the beginning. Also, if you do select the first approach, consider provisioning for switching to the second method by reserving a sub-version number. For example, for an application version in the form `2.3.4` you can make the object model version to be in the form `0x0203040000`, reserving the last two bytes for a sub-version. Later on you can use it to switch to the version per feature approach.

13.2 Schema Migration

Once we enable schema evolution by specifying the object model version, in addition to the schema creation statements, the ODB compiler starts generating schema migration statements for each version all the way from the base to the current. As with schema creation, schema migration can be generated either as a set of SQL files or embedded into the generated C++ code (`--schema-format` option).

For each migration step, that is from one version to the next, ODB generates two sets of statements: pre-migration and post-migration. The pre-migration statements *"relax"* the database schema so that both old and new data can co-exist. At this stage new columns and tables are added while old constraints are dropped. The post-migration statements *"tighten"* the database schema back so that only data conforming to the new format can remain. At this stage old columns and tables are dropped and new constraints are added. Now you can probably guess where the data migration fits into this — between the pre and post schema migrations where we can both access the old data and create the new one.

If the schema is being generated as standalone SQL files, then we end up with a pair of files for each step: the pre-migration file and the post-migration file. For the `person` example we started in the previous section we will have the `person-002-pre.sql` and `person-002-post.sql` files. Here `002` is the version *to* which we are migrating while the `pre` and `post` suffixes specify the migration stage. So if we wanted to migrate a `person` database from version `1` to `2`, then we would first execute `person-002-pre.sql`, then migrate the data, if any (discussed in more detail in the next section), and finally execute `person-002-post.sql`. If our database is several versions behind, for example the database has version `1` while the current version is `5`, then we simply perform this set of steps for each

version until we reach the current version.

If we look at the contents of the `person-002-pre.sql` file, we will see the following (or equivalent, depending on the database used) statement:

```
ALTER TABLE "person"
  ADD COLUMN "middle" TEXT NULL;
```

As we would expect, this statement adds a new column corresponding to the new data member. An observant reader would notice, however, that the column is added as `NULL` even though we never requested this semantics in our object model. Why is the column added as `NULL`? If during migration the `person` table already contains rows (that is, existing objects), then an attempt to add a non-`NULL` column that doesn't have a default value will fail. As a result, ODB will initially add a new column that doesn't have a default value as `NULL` but then clean this up at the post-migration stage. This way your data migration code is given a chance to assign some meaningful values for the new data member for all the existing objects. Here are the contents of the `person-002-post.sql` file:

```
ALTER TABLE "person"
  ALTER COLUMN "middle" SET NOT NULL;
```

Currently ODB directly supports the following elementary database schema changes:

- add table
- drop table
- add column
- drop column
- alter column, set `NULL`/`NOT NULL`
- add foreign key
- drop foreign key
- add index
- drop index

More complex changes can normally be implemented in terms of these building blocks. For example, to change a type of a data member (which leads to a change of a column type), we can add a new data member with the desired type (add column), migrate the data, and then delete the old data member (drop column). ODB will issue diagnostics for cases that are currently not supported directly. Note also that some database systems (notably SQLite) have a number of limitations in their support for schema changes. For more information on these database-specific limitations, refer to the "Limitations" sections in Part II, "Database Systems".

How do we know what the current database version is? That is, the version *from* which we need to migrate? We need to know this, for example, in order to determine the set of migrations we have to perform. By default, when schema evolution is enabled, ODB maintains this information

in a special table called `schema_version` that has the following (or equivalent, depending on the database used) definition:

```
CREATE TABLE "schema_version" (
  "name" TEXT NOT NULL PRIMARY KEY,
  "version" BIGINT NOT NULL,
  "migration" BOOLEAN NOT NULL);
```

The `name` column is the schema name as specified with the `--schema-name` option. It is empty for the default schema. The `version` column contains the current database version. And, finally, the `migration` flag indicates whether we are in the process of migrating the database, that is, between the pre and post-migration stages.

The schema creation statements (`person.sql` in our case) create this table and populate it with the initial model version. For example, if we executed `person.sql` corresponding to version `1` of our object model, then `name` would have been empty (which signifies the default schema since we didn't specify `--schema-name`), `version` will be `1` and `migration` will be `FALSE`.

The pre-migration statements update the version and set the migration flag to `TRUE`. Continuing with our example, after executing `person-002-pre.sql`, `version` will become `2` and `migration` will be set to `TRUE`. The post-migration statements simply clear the migration flag. In our case, after running `person-002-post.sql`, `version` will remain `2` while `migration` will be reset to `FALSE`.

Note also that above we mentioned that the schema creation statements (`person.sql`) create the `schema_version` table. This means that if we enable schema evolution support in the middle of a project, then we could already have existing databases that don't include this table. As a result, ODB will not be able to handle migrations for such databases unless we manually add the `schema_version` table and populate it with the correct version information. For this reason, it is highly recommended that you consider whether to use schema evolution and, if so, enable it from the beginning of your project.

The `odb::database` class provides an API for accessing and modifying the current database version:

```
namespace odb
{
  using schema_version = unsigned long long;

  struct LIBODB_EXPORT schema_version_migration
  {
    schema_version_migration (schema_version = 0,
                              bool migration = false);

    schema_version version;
    bool migration;
```

```
    // This class also provides the ==, !=, <, >, <=, and >= operators.
    // Version ordering is as follows: {1,f} < {2,t} < {2,f} < {3,t}.
  };

  class database
  {
  public:
    ...

    schema_version
    schema_version (const std::string& name = "") const;

    bool
    schema_migration (const std::string& name = "") const;

    const schema_version_migration&
    schema_version_migration (const std::string& name = "") const;

    // Set schema version and migration state manually.
    //
    void
    schema_version_migration (schema_version,
                              bool migration,
                              const std::string& name = "");

    void
    schema_version_migration (const schema_version_migration&,
                              const std::string& name = "");

    // Set default schema version table for all schemas.
    //
    void
    schema_version_table (const std::string& table_name);

    // Set schema version table for a specific schema.
    //
    void
    schema_version_table (const std::string& table_name,
                          const std::string& name);
  };
}
```

The `schema_version()` and `schema_migration()` accessors return the current database version and migration flag, respectively. The optional `name` argument is the schema name. If the database schema hasn't been created (that is, there is no corresponding entry in the `schema_version` table or this table does not exist), then `schema_version()` returns `0`. The `schema_version_migration()` accessor returns both version and migration flag together in the `schema_version_migration` `struct`.

You may already have a version table in your database or you (or your database administrator) may prefer to keep track of versions your own way. You can instruct ODB not to create the `schema_version` table with the `--suppress-schema-version` option. However, ODB still needs to know the current database version in order for certain schema evolution mechanisms to function properly. As a result, in this case, you will need to set the schema version on the database instance manually using the schema_version_migration() modifier. Note that the modifier API is not thread-safe. That is, you should not modify the schema version while other threads may be accessing or modifying the same information.

Note also that the accessors we discussed above will only query the `schema_version` table once and, if the version could be determined, cache the result. If, however, the version could not be determined (that is, `schema_version()` returned 0), then a subsequent call will re-query the table. While it is probably a bad idea to modify the database schema while the application is running (other than via the `schema_catalog` API, as discussed below), if for some reason you need ODB to re-query the version, then you can manually set it to 0 using the `schema_version_migration()` modifier.

It is also possible to change the name of the table that stores the schema version using the `--schema-version-table` option. You will also need to specify this alternative name on the `database` instance using the `schema_version_table()` modifier. The first version specifies the default table that is used for all the schema names. The second version specifies the table for a specific schema. The table name should be database-quoted, if necessary.

If we are generating our schema migrations as standalone SQL files, then the migration workflow could look like this:

1. The database administrator determines the current database version. If migration is required, then for each migration step (that is, from one version to the next), they perform the following:
2. Execute the pre-migration file.
3. Execute our application (or a separate migration program) to perform data migration (discussed later). Our application can determine that is is being executed in the "migration mode" by calling `schema_migration()` and then which migration code to run by calling `schema_version()`.
4. Execute the post-migration file.

These steps become more integrated and automatic if we embed the schema creation and migration code into the generated C++ code. Now we can perform schema creation, schema migration, and data migration as well as determine when each step is necessary programmatically from within the application.

Schema evolution support adds the following extra functions to the `odb::schema_catalog` class, which we first discussed in Section 3.4, "Database".

```
namespace odb
{
  class schema_catalog
  {
  public:
    ...


    // Schema migration.
    //
    static void
    migrate_schema_pre (database&,
                        schema_version,
                        const std::string& name = "");

    static void
    migrate_schema_post (database&,
                         schema_version,
                         const std::string& name = "");

    static void
    migrate_schema (database&,
                    schema_version,
                    const std::string& name = "");

    // Data migration.
    //
    // Discussed in the next section.


    // Combined schema and data migration.
    //
    static void
    migrate (database&,
             schema_version = 0,
             const std::string& name = "");

    // Schema version information.
    //
    static schema_version
    base_version (const database&,
                  const std::string& name = "");

    static schema_version
    base_version (database_id,
                  const std::string& name = "");

    static schema_version
```

```
    current_version (const database&,
                     const std::string& name = "");

    static schema_version
    current_version (database_id,
                     const std::string& name = "");

    static schema_version
    next_version (const database&,
                  schema_version = 0,
                  const std::string& name = "");

    static schema_version
    next_version (database_id,
                  schema_version,
                  const std::string& name = "");
  };
}
```

The `migrate_schema_pre()` and `migrate_schema_post()` static functions perform a single stage (that is, pre or post) of a single migration step (that is, from one version to the next). The `version` argument specifies the version we are migrating to. For instance, in our `person` example, if we know that the database version is `1` and the next version is `2`, then we can execute code like this:

```
transaction t (db.begin ());

schema_catalog::migrate_schema_pre (db, 2);

// Data migration goes here.

schema_catalog::migrate_schema_post (db, 2);

t.commit ();
```

If you don't have any data migration code to run, then you can perform both stages with a single call using the `migrate_schema()` static function.

The `migrate()` static function perform both schema and data migration (we discuss data migration in the next section). It can also perform several migration steps at once. If we don't specify its target version, then it will migrate (if necessary) all the way to the current model version. As an extra convenience, `migrate()` will also create the database schema if none exists. As a result, if we don't have any data migration code or we have registered it with `schema_catalog` (as discussed later), then the database schema creation and migration, whichever is necessary, if at all, can be performed with a single function call:

```
transaction t (db.begin ());
schema_catalog::migrate (db);
t.commit ();
```

Note also that `schema_catalog` is integrated with the `odb::database` schema version API. In particular, `schema_catalog` functions will query and synchronize the schema version on the `database` instance if and when required.

The `schema_catalog` class also allows you to iterate over known versions (remember, there could be "gaps" in version numbers) with the `base_version()`, `current_version()` and `next_version()` static functions. The `base_version()` and `current_version()` functions return the base and current object model versions, respectively. That is, the lowest version from which we can migrate and the version that we ultimately want to migrate to. The `next_version()` function returns the next known version. If the passed version is greater or equal to the current version, then this function will return the current version plus one (that is, one past current). If we don't specify the version, then `next_version()` will use the current database version as the starting point. Note also that the schema version information provided by these functions is only available if we embed the schema migration code into the generated C++ code. For standalone SQL file migrations this information is normally not needed since the migration process is directed by an external entity, such as a database administrator or a script.

Most `schema_catalog` functions presented above also accept the optional schema name argument. If the passed schema name is not found, then the `odb::unknown_schema` exception is thrown. Similarly, functions that accept the schema version argument will throw the `odb::unknown_schema_version` exception if the passed or current version is invalid. Refer to Section 3.14, "ODB Exceptions" for more information on these exceptions.

To illustrate how all these parts fit together, consider the following more realistic database schema management example. Here we want to handle the schema creation in a special way and perform each migration step in its own transaction.

```
schema_version v (db.schema_version ());
schema_version bv (schema_catalog::base_version (db));
schema_version cv (schema_catalog::current_version (db));

if (v == 0)
{
  // No schema in the database. Create the schema and
  // initialize the database.
  //
  transaction t (db.begin ());
  schema_catalog::create_schema (db);

  // Populate the database with initial data, if any.

  t.commit ();
}
```

```
else if (v < cv)
{
  // Old schema (and data) in the database, migrate them.
  //

  if (v < bv)
  {
    // Error: migration from this version is no longer supported.
  }

  for (v = schema_catalog::next_version (db, v);
       v <= cv;
       v = schema_catalog::next_version (db, v))
  {
    transaction t (db.begin ());
    schema_catalog::migrate_schema_pre (db, v);

    // Data migration goes here.

    schema_catalog::migrate_schema_post (db, v);
    t.commit ();
  }
}
else if (v > cv)
{
  // Error: old application trying to access new database.
}
```

13.3 Data Migration

In quite a few cases specifying the default value for new data members will be all that's required to handle the existing objects. For example, the natural default value for the new middle name that we have added is an empty string. And we can handle this case with the `db default` pragma and without any extra C++ code:

```
#pragma db model version(1, 2)

#pragma db object
class person
{
  ...


  #pragma db default("")
  std::string middle_;
};
```

However, there will be situations where we would need to perform more elaborate data migrations, that is, convert old data to the new format. As an example, suppose we want to add gender to our `person` class. And, instead of leaving it unassigned for all the existing objects, we will try to guess it from the first name. This is not particularly accurate but it could be sufficient for our hypothetical application:

```
#pragma db model version(1, 3)

enum gender {male, female};

#pragma db object
class person
{
  ...

  gender gender_;
};
```

As we have discussed earlier, there are two ways to perform data migration: immediate and gradual. To recap, with immediate migration we migrate all the existing objects at once, normally after the schema pre-migration statements but before the post-migration statements. With gradual migration, we make sure the new object model can accommodate both old and new data and gradually migrate existing objects as the application runs and the opportunities to do so arise, for example, an object is updated.

There is also another option for data migration that is not discussed further in this section. Instead of using our C++ object model we could execute ad-hoc SQL statements that perform the necessary conversions and migrations directly on the database server. While in certain cases this can be a better option from the performance point of view, this approach is often limited in terms of the migration logic that we can handle.

13.3.1 Immediate Data Migration

Let's first see how we can implement an immediate migration for the new `gender_` data member we have added above. If we are using standalone SQL files for migration, then we could add code along these lines somewhere early in `main()`, before the main application logic:

```
int
main ()
{
  ...

  odb::database& db = ...

  // Migrate data if necessary.
  //
  if (db.schema_migration ())
```

```
  {
    switch (db.schema_version ())
    {
    case 3:
      {
        // Assign gender to all the existing objects.
        //
        transaction t (db.begin ());

        for (person& p: db.query<person> ())
        {
          p.gender (guess_gender (p.first ()));
          db.update (p);
        }

        t.commit ();
        break;
      }
    }
  }

  ...
}
```

If you have a large number of objects to migrate, it may also be a good idea, from the performance point of view, to break one big transaction that we now have into multiple smaller transactions (Section 3.5, "Transactions"). For example:

```
case 3:
  {
    transaction t (db.begin ());

    size_t n (0);
    for (person& p: db.query<person> ())
    {
      p.gender (guess_gender (p.first ()));
      db.update (p);

      // Commit the current transaction and start a new one after
      // every 100 updates.
      //
      if (n++ % 100 == 0)
      {
        t.commit ();
        t.reset (db.begin ());
      }
    }

    t.commit ();
    break;
  }
```

While it looks straightforward enough, as we add more migration snippets, this approach can quickly become unmaintainable. Instead of having all the migrations in a single function and determining when to run each piece ourselves, we can package each migration into a separate function, register it with the `schema_catalog` class, and let ODB figure out when to run which migration functions. To support this functionality, `schema_catalog` provides the following data migration API:

```
namespace odb
{
  class schema_catalog
  {
  public:
    ...

    // Data migration.
    //
    static std::size_t
    migrate_data (database&,
                  schema_version = 0,
                  const std::string& name = "");

    using data_migration_function_type = void (database&);

    // Common (for all the databases) data migration, C++98/03 version:
    //
    template <schema_version v, schema_version base>
    static void
    data_migration_function (data_migration_function_type*,
                             const std::string& name = "");

    // Common (for all the databases) data migration, C++11 version:
    //
    template <schema_version v, schema_version base>
    static void
    data_migration_function (std::function<data_migration_function_type>,
                             const std::string& name = "");

    // Database-specific data migration, C++98/03 version:
    //
    template <schema_version v, schema_version base>
    static void
    data_migration_function (database&,
                             data_migration_function_type*,
                             const std::string& name = "");

    template <schema_version v, schema_version base>
    static void
    data_migration_function (database_id,
                             data_migration_function_type*,
                             const std::string& name = "");
```

```
    // Database-specific data migration, C++11 version:
    //
    template <schema_version v, schema_version base>
    static void
    data_migration_function (database&,
                             std::function<data_migration_function_type>,
                             const std::string& name = "");

    template <schema_version v, schema_version base>
    static void
    data_migration_function (database_id,
                             std::function<data_migration_function_type>,
                             const std::string& name = "");
  };

  // Static data migration function registration, C++98/03 version:
  //
  template <schema_version v, schema_version base>
  struct data_migration_entry
  {
    data_migration_entry (data_migration_function_type*,
                          const std::string& name = "");

    data_migration_entry (database_id,
                          data_migration_function_type*,
                          const std::string& name = "");
  };

  // Static data migration function registration, C++11 version:
  //
  template <schema_version v, schema_version base>
  struct data_migration_entry
  {
    data_migration_entry (std::function<data_migration_function_type>,
                          const std::string& name = "");

    data_migration_entry (database_id,
                          std::function<data_migration_function_type>,
                          const std::string& name = "");
  };
}
```

The `migrate_data()` static function performs data migration for the specified version. If no version is specified, then it will use the current database version and also check whether the database is in migration, that is, `database::schema_migration()` returns `true`. As a result, all we need to do in our `main()` is call this function. It will check if migration is required and if so, call all the migration functions registered for this version. For example:

```
int
main ()
{
  ...

  database& db = ...

  // Check if we need to migrate any data and do so
  // if that's the case.
  //
  schema_catalog::migrate_data (db);

  ...
}
```

The `migrate_data()` function returns the number of migration functions called. You can use this value for debugging or logging.

The only other step that we need to perform is register our data migration functions with `schema_catalog`. At the lower level we can call the `data_migration_function()` static function for every migration function we have, for example, at the beginning of `main()`. For each version, data migration functions are called in the order of registration.

A more convenient approach, however, is to use the `data_migration_entry` helper class template to register the migration functions during static initialization. This way we can keep the migration function and its registration code next to each other. Here is how we can reimplement our `gender` migration code to use this mechanism:

```
static void
migrate_gender (odb::database& db)
{
  transaction t (db.begin ());

  for (person& p: db.query<person> ())
  {
    p.gender (guess_gender (p.first ()));
    db.update (p);
  }

  t.commit ();
}

static const odb::data_migration_entry<3, MYAPP_BASE_VERSION>
migrate_gender_entry (&migrate_gender);
```

The first template argument to the `data_migration_entry` class template is the version we want this data migration function to be called for. The second template argument is the base model version. This second argument is necessary to detect the situation where we no longer need

this data migration function. Remember that when we move the base model version forward, migrations from any version below the new base are no longer possible. We, however, may still have migration functions registered for those lower versions. Since these functions will never be called, they are effectively dead code and it would be useful to identify and remove them. To assist with this, `data_migration_entry` (and lower lever `data_migration_function()`) will check at compile time (that is, `static_assert`) that the registration version is greater than the base model version.

In the above example we use the `MYAPP_BASE_VERSION` macro that is presumably defined in a central place, for example, `version.hxx`. This is the recommended approach since we can update the base version in a single place and have the C++ compiler automatically identify all the data migration functions that can be removed.

In C++11 we can also create a template alias so that we don't have to repeat the base model macro in every registration, for example:

```
template <schema_version v>
using migration_entry = odb::data_migration_entry<v, MYAPP_BASE_VERSION>;

static const migration_entry<3>
migrate_gender_entry (&migrate_gender);
```

For cases where you need to by-pass the base version check, for example, to implement your own registration helper, ODB also provides "unsafe" versions of the `data_migration_function()` functions that take the version as a function argument rather than as a template parameter.

In C++11 we can also use lambdas as migration functions, which makes the migration code more concise:

```
static const migration_entry<3>
migrate_gender_entry (
  [] (odb::database& db)
  {
    transaction t (db.begin ());

    for (person& p: db.query<person> ())
    {
      p.gender (guess_gender (p.first ()));
      db.update (p);
    }

    t.commit ();
  });
```

If we are using embedded schema migrations, then both schema and data migration is integrated and can be performed with a single call to the `schema_catalog::migrate()` function that we discussed earlier. For example:

```
int
main ()
{
  ...

  database& db = ...

  // Check if we need to migrate the database and do so
  // if that's the case.
  //
  {
    transaction t (db.begin ());
    schema_catalog::migrate (db);
    t.commit ();
  }

  ...
}
```

Note, however, that in this case we call `migrate()` within a transaction (for the schema migration part) which means that our migration functions will also be called within this transaction. As a result, we will need to adjust our migration functions not to start their own transaction:

```
static void
migrate_gender (odb::database& db)
{
  // Assume we are already in a transaction.
  //
  for (person& p: db.query<person> ())
  {
    p.gender (guess_gender (p.first ()));
    db.update (p);
  }
}
```

If, however, we want more granular transactions, then we can use the lower-level `schema_catalog` functions to gain more control, as we have seen at the end of the previous section. Here is the relevant part of that example with an added data migration call:

```
  // Old schema (and data) in the database, migrate them.
  //
  for (v = schema_catalog::next_version (db, v);
       v <= cv;
       v = schema_catalog::next_version (db, v))
  {
```

```
    transaction t (db.begin ());
    schema_catalog::migrate_schema_pre (db, v);
    schema_catalog::migrate_data (db, v);
    schema_catalog::migrate_schema_post (db, v);
    t.commit ();
  }
```

13.3.2 Gradual Data Migration

If the number of existing objects that require migration is large, then an all-at-once, immediate migration, while simple, may not be practical from a performance point of view. In this case, we can perform a gradual migration as the application does its normal functions.

With gradual migrations, the object model must be capable of representing data that conforms to both old and new formats at the same time since, in general, the database will contain a mixture of old and new objects. For example, in case of our `gender` data member, we need a special value that represents the "no gender assigned yet" case (an old object). We also need to assign this special value to all the existing objects during the schema pre-migration stage. One way to do this would be add a special value to our `gender` enum and then make it the default value with the `db default` pragma. A cleaner and easier approach, however, is to use `NULL` as a special value. We can add support for the `NULL` value semantics to any existing type by wrapping it with `odb::nullable`, `boost::optional` or similar (Section 7.3, "Pointers and `NULL` Value Semantics"). We also don't need to specify the default value explicitly since `NULL` is used automatically. Here is how we can use this approach in our `gender` example:

```
#include <odb/nullable.hxx>

#pragma db object
class person
{
  ...

  odb::nullable<gender> gender_;
};
```

A variety of strategies can be employed to implement gradual migrations. For example, we can migrate the data when the object is updated as part of the normal application logic. While there is no migration cost associated with this approach (the object is updated anyway), depending on how often objects are typically updated, this strategy can take a long time to complete. An alternative strategy would be to perform an update whenever an old object is loaded. Yet another strategy is to have a separate thread that slowly migrates all the old objects as the application runs.

As an example, let us implement the first approach for our `gender` migration. While we could have added the necessary code throughout the application, from the maintenance point of view, it is best to try and localize the gradual migration logic to the persistent classes that it affects. And for this database operation callbacks (Section 14.1.7, "`callback`") are a very useful mechanism. In our case, all we have to do is handle the `post_load` event where we guess the gender if it is `NULL`:

```
#include <odb/core.hxx>     // odb::database
#include <odb/callback.hxx> // odb::callback_event
#include <odb/nullable.hxx>

#pragma db object callback(migrate)
class person
{
  ...

  void
  migrate (odb::callback_event e, odb::database&)
  {
    if (e == odb::callback_event::post_load)
    {
      // Guess gender if not assigned.
      //
      if (gender_.null ())
        gender_ = guess_gender (first_);
    }
  }

  odb::nullable<gender> gender_;
};
```

In particular, we don't have to touch any of the accessors or modifiers or the application logic — all of them can assume that the value can never be `NULL`. And when the object is next updated, the new `gender` value will be stored automatically.

All gradual migrations normally end up with a terminating immediate migration some number of versions down the line, when the bulk of the objects has presumably been converted. This way we don't have to keep the gradual migration code around forever. Here is how we could implement a terminating migration for our example:

```
// person.hxx
//
#pragma db model version(1, 4)

#pragma db object
class person
{
  ...
```

```
  gender gender_;
};

// person.cxx
//
static void
migrate_gender (odb::database& db)
{
  using query = odb::query<person>;

  for (person& p: db.query<person> (query::gender.is_null ()))
  {
    p.gender (guess_gender (p.first ()));
    db.update (p);
  }
}

static const odb::data_migration_entry<4, MYAPP_BASE_VERSION>
migrate_gender_entry (&migrate_gender);
```

A couple of points to note about this code. Firstly, we removed all the gradual migration logic (the callback) from the class and replaced it with the immediate migration function. We also removed the `odb::nullable` wrapper (and therefore disallowed the `NULL` values) since after this migration all the objects will have been converted. Finally, in the migration function, we only query the database for objects that need migration, that is, have `NULL` gender.

13.4 Soft Object Model Changes

Let us consider another common kind of object model change: we delete an old member, add a new one, and need to copy the data from the old to the new, perhaps applying some conversion. For example, we may realize that in our application it is a better idea to store a person's name as a single string rather than split it into three fields. So what we would like to do is add a new data member, let's call it `name_`, convert all the existing split names, and then delete the `first_`, `middle_`, and `last_` data members.

While this sounds straightforward, there is a problem. If we delete (that is, physically remove from the source code) the old data members, then we won't be able to access the old data. The data will still be available in the database between the schema pre and post-migrations, it is just we will no longer be able to access it through our object model. And if we keep the old data members around, then the old data will remain stored in the database even after the schema post-migration.

There is also a more subtle problem that has to do with existing migrations for the previous versions. Remember, in version `3` of our `person` example we added the `gender_` data member. We also have a data migration function which guesses the gender based on the first name. Deleting the `first_` data member from our class will obviously break this code. But even

adding the new `name_` data member will cause problems because when we try to update the object in order to store the new gender, ODB will try to update `name_` as well. But there is no corresponding column in the database yet. When we run this migration function, we are still several versions away from the point where the `name` column will be added.

This is a very subtle but also very important implication to understand. Unlike the main application logic, which only needs to deal with the current model version, data migration code works on databases that can be multiple versions behind the current version.

How can we resolve this problem? It appears what we need is the ability to add or delete data members starting from a specific version. In ODB this mechanism is called soft member additions and deletions. A soft-added member is only treated as persistent starting from the addition version. A soft-deleted member is persistent until the deletion version (but including the migration stage). In its essence, soft model changes allow us to maintain multiple versions of our object model all with a single set of persistent classes. Let us now see how this functionality can help implement our changes:

```
#pragma db model version(1, 4)

#pragma db object
class person
{
  ...

  #pragma db id auto
  unsigned long long id_;

  #pragma db deleted(4)
  std::string first_;

  #pragma db deleted(4)
  std::string middle_;

  #pragma db deleted(4)
  std::string last_;

  #pragma db added(4)
  std::string name_;

  gender gender_;
};
```

The migration function for this change could then look like this:

```
static void
migrate_name (odb::database& db)
{
  for (person& p: db.query<person> ())
```

```
  {
    p.name (p.first () + " " +
            p.middle () + (p.middle ().empty () ? "" : " ") +
            p.last ());
    db.update (p);
  }
}

static const odb::data_migration_entry<4, MYAPP_BASE_VERSION>
migrate_name_entry (&migrate_name);
```

Note also that no changes are required to the gender migration function.

As you may have noticed, in the code above we assumed that the `person` class still provides public accessors for the now deleted data members. This might not be ideal since now they should not be used by the application logic. The only code that may still need to access them is the migration functions. The recommended way to resolve this is to remove the accessors/modifiers corresponding to the deleted data member, make migration functions static functions of the class being migrated, and then access the deleted data members directly. For example:

```
#pragma db model version(1, 4)

#pragma db object
class person
{
  ...

private:
  friend class odb::access;

  #pragma db id auto
  unsigned long long id_;

  #pragma db deleted(4)
  std::string first_;

  #pragma db deleted(4)
  std::string middle_;

  #pragma db deleted(4)
  std::string last_;

  #pragma db added(4)
  std::string name_;

  gender gender_;

private:
  static void
  migrate_gender (odb::database&);
```

```
  static void
  migrate_name (odb::database&);
};

void person::
migrate_gender (odb::database& db)
{
  for (person& p: db.query<person> ())
  {
    p.gender_ = guess_gender (p.first_);
    db.update (p);
  }
}

static const odb::data_migration_entry<3, MYAPP_BASE_VERSION>
migrate_name_entry (&migrate_gender);

void person::
migrate_name (odb::database& db)
{
  for (person& p: db.query<person> ())
  {
    p.name_ = p.first_ + " " +
              p.middle_ + (p.middle_.empty () ? "" : " ") +
              p.last_;
    db.update (p);
  }
}

static const odb::data_migration_entry<4, MYAPP_BASE_VERSION>
migrate_name_entry (&migrate_name);
```

Another potential issue with the soft-deletion is the requirement to keep the delete data members in the class. While they will not be initialized in the normal operation of the application (that is, not a migration), this can still be a problem if we need to minimize the memory footprint of our classes. For example, we may cache a large number of objects in memory and having three `std::string` data members can be a significant overhead.

The recommended way to resolve this issue is to place all the deleted data members into a dynamically allocated composite value type. For example:

```
#pragma db model version(1, 4)

#pragma db object
class person
{
  ...

  #pragma db id auto
```

```
  unsigned long long id_;

  #pragma db added(4)
  std::string name_;

  gender gender_;

  #pragma db value
  struct deleted_data
  {
    #pragma db deleted(4)
    std::string first_;

    #pragma db deleted(4)
    std::string middle_;

    #pragma db deleted(4)
    std::string last_;
  };

  #pragma db column("")
  std::unique_ptr<deleted_data> dd_;

  ...
};
```

ODB will then automatically allocate the deleted value type if any of the deleted data members are being loaded. During the normal operation, however, the pointer will stay `NULL` and therefore reduce the common case overhead to a single pointer per class. Note that we make the composite value column prefix empty (the `db column("")` pragma) in order to keep the same column names for the deleted data members.

Soft-added and deleted data members can be used in objects, composite values, views, and container value types. We can also soft-add and delete data members of simple, composite, pointer to object, and container types. Only special data members, such as the object id and the optimistic concurrency version, cannot be soft-added or deleted.

It is also possible to soft-delete a persistent class. We can still work with the existing objects of such a class, however, no table is created in new databases for soft-deleted classes. To put it another way, a soft-delete class is like an abstract class (no table) but which can still be loaded, updated, etc. Soft-added persistent classes do not make much sense and are therefore not supported.

As an example of a soft-deleted class, suppose we want to replace our `person` class with the new `employee` object and migrate the data. Here is how we could do this:

```
#pragma db model version(1, 5)

#pragma db object deleted(5)
class person
{
  ...
};

#pragma db object
class employee
{
  ...

  #pragma db id auto
  unsigned long long id_;

  std::string name_;
  gender gender_;

  static void
  migrate_person (odb::database&);
};

void employee::
migrate_person (odb::database& db)
{
  for (person& p: db.query<person> ())
  {
    employee e (p.name (), p.gender ());
    db.persist (e);
  }
}

static const odb::data_migration_entry<5, MYAPP_BASE_VERSION>
migrate_person_entry (&migrate_person);
```

As we have seen above, hard member additions and deletions can (and most likely will) break existing data migration code. Why, then, not treat all the changes, or at least additions, as soft? ODB requires you to explicitly request this semantics because support for soft-added and deleted data members incurs runtime overhead. And there can be plenty of cases where there is no existing data migration and therefore hard additions and deletions are sufficient.

In some cases a hard addition or deletion will result in a compile-time error. For example, one of the data migration functions may reference the data member we just deleted. In many cases, however, such errors can only be detected at runtime, and, worse yet, only when the migration function is executed. For example, we may hard-add a new data member that an existing migration function will try to indirectly store in the database as part of an object update. As a result, it is highly recommended that you always test your application with the database that starts at the base version so that every data migration function is called and therefore ensured to still work

correctly.

To help with this problem you can also instruct ODB to warn you about any hard additions or deletions with the `--warn-hard-add`, `--warn-hard-delete`, and `--warn-hard` command line options. ODB will only warn you about hard changes in the current version and only for as long as it is open, which makes this mechanism fairly usable.

You may also be wondering why we have to specify the addition and deletion versions explicitly. It may seem like the ODB compiler should be able to figure this out automatically. While it is theoretically possible, to achieve this, ODB would have to also maintain a separate changelog of the C++ object model in addition to the database schema changelog it already maintains. While being a lot more complex, such an additional changelog would also complicate the workflow significantly. In this light, maintaining this change information as part of the original source files appears to be a cleaner and simpler approach.

As we discussed before, when we move the base model version forward we essentially drop support for migrations from versions before the new base. As a result, it is no longer necessary to maintain the soft semantics of additions and deletions up to and including the new base version. ODB will issue diagnostics for all such members and classes. For soft deletions we can simply remove the data member or class entirely. For soft additions we only need to remove the `db added` pragma.

13.4.1 Reuse Inheritance Changes

Besides adding and deleting data members, another way to alter the object's table is using reuse-style inheritance. If we add a new reuse base, then, from the database schema point of view, this is equivalent to adding all its columns to the derived object's table. Similarly, deleting reuse inheritance results in all the base's columns being deleted from the derived's table.

In the future ODB may provide direct support for soft addition and deletion of inheritance. Currently, however, this semantics can be emulated with soft-added and deleted data members. The following table describes the most common scenarios depending on where columns are added or deleted, that is, base table, derived table, or both.

DELETE	HARD	SOFT
In both (delete inheritance and base)	Delete inheritance and base. Move object id to derived.	Soft-delete base. Mark all data members (except id) in base as soft-deleted.
In base only (delete base)	Option 1: mark base as abstract. Option 2: move all the base member to derived, delete base.	Soft-delete base.
In derived only (delete inheritance)	Delete inheritance, add object id to derived.	Option 1: copy base to a new soft-deleted base, inherit from it instead. Mark all the data members (expect id) in this new base as soft-deleted. Note: we add the new base as soft-deleted to get notified when we can remove it. Option 2: Copy all the data members from base to derived and mark them as soft-deleted in derived.

ADD	HARD	SOFT
In both (add new base and inheritance)	Add new base and inheritance. Potentially move object id member from derived to base.	Add new base and mark all its data members as soft-added. Add inheritance. Move object id from derived to base.
In base only (refactor existing data to new base)	Add new base and move data members from derived to base. Note: in most cases the new base will be made abstract which make this scenario non-schema changing.	The same as HARD.
In derived only (add inheritance to existing base)	Add inheritance, delete object id in derived.	Copy existing base to a new abstract base and inherit from it. Mark all the database members in the new base as soft-added (except object id). When notified by the ODB compiler that the soft addition of the data members is no longer necessary, delete the copy and inherit from the original base.

13.4.2 Polymorphism Inheritance Changes

Unlike reuse inheritance, adding or deleting a polymorphic base does not result in the base's data members being added or deleted from the derived object's table because each class in a polymorphic hierarchy is stored in a separate table. There are, however, other complications due to the presence of special columns (discriminator in the root table and object id links in derived tables) which makes altering the hierarchy structure difficult to handle automatically. Adding or deleting (including soft-deleting) of leaf classes (or leaf sub-hierarchies) in a polymorphic hierarchy is fully supported. Any more complex changes, such as adding or deleting the root or an intermediate base or getting an existing class into or out of a polymorphic hierarchy can be handled by creating a new leaf class (or leaf sub-hierarchy), soft-deleting the old class, and migrating the data.

14 ODB Pragma Language

As we have already seen in previous chapters, ODB uses a pragma-based language to capture database-specific information about C++ types. This chapter describes the ODB pragma language in more detail. It can be read together with other chapters in the manual to get a sense of what kind of configurations and mapping fine-tuning are possible. You can also use this chapter as a reference at a later stage.

An ODB pragma has the following syntax:

```
#pragma db qualifier [specifier specifier ...]
```

The *qualifier* tells the ODB compiler what kind of C++ construct this pragma describes. Valid qualifiers are `object`, `view`, `value`, `member`, `namespace`, `model`, `index`, and `map`. A pragma with the `object` qualifier describes a persistent object type. It tells the ODB compiler that the C++ class it describes is a persistent class. Similarly, pragmas with the `view` qualifier describe view types, the `value` qualifier describes value types and the `member` qualifier is used to describe data members of persistent object, view, and value types. The `namespace` qualifier is used to describe common properties of objects, views, and value types that belong to a C++ namespace while the `model` qualifier describes the whole C++ object model. The `index` qualifier defines a database index. And, finally, the `map` qualifier describes a mapping between additional database types and types for which ODB provides built-in support.

The *specifier* informs the ODB compiler about a particular database-related property of the C++ declaration. For example, the `id` member specifier tells the ODB compiler that this member contains this object's identifier. Below is the declaration of the `person` class that shows how we can use ODB pragmas:

```
#pragma db object
class person
{
  ...
private:
  #pragma db member id
  unsigned long long id_;
  ...
};
```

In the above example we don't explicitly specify which C++ class or data member the pragma belongs to. Rather, the pragma applies to a C++ declaration that immediately follows the pragma. Such pragmas are called *positioned pragmas*. In positioned pragmas that apply to data members, the `member` qualifier can be omitted for brevity, for example:

```
  #pragma db id
  unsigned long long id_;
```

Note also that if the C++ declaration immediately following a position pragma is incompatible with the pragma qualifier, an error will be issued. For example:

```
  #pragma db object  // Error: expected class instead of data member.
  unsigned long long id_;
```

While keeping the C++ declarations and database declarations close together eases maintenance and increases readability, we can also place them in different parts of the same header file or even factor them to a separate file. To achieve this we use the so called *named pragmas*. Unlike positioned pragmas, named pragmas explicitly specify the C++ declaration to which they apply by adding the declaration name after the pragma qualifier. For example:

```
class person
{
  ...
private:
  unsigned long long id_;
  ...
};

#pragma db object(person)
#pragma db member(person::id_) id
```

Note that in the named pragmas for data members the `member` qualifier is no longer optional. The C++ declaration name in the named pragmas is resolved using the standard C++ name resolution rules, for example:

```
namespace db
{
  class person
  {
    ...
  private:
    unsigned long long id_;
    ...
  };
}

namespace db
{
  #pragma db object(person)  // Resolves db::person.
}

#pragma db member(db::person::id_) id
```

As another example, the following code fragment shows how to use the named value type pragma to map a C++ type to a native database type:

```
#pragma db value(bool) type("INT")

#pragma db object
class person
{
  ...
private:
  bool married_; // Mapped to INT NOT NULL database type.
  ...
};
```

If we would like to factor the ODB pragmas into a separate file, we can include this file into the original header file (the one that defines the persistent types) using the `#include` directive, for example:

```
// person.hxx

class person
{
  ...
};

#ifdef ODB_COMPILER
#  include "person-pragmas.hxx"
#endif
```

Alternatively, instead of using the `#include` directive, we can use the `--odb-epilogue` option to make the pragmas known to the ODB compiler when compiling the original header file, for example:

```
--odb-epilogue  '#include "person-pragmas.hxx"'
```

The following sections cover the specifiers applicable to all the qualifiers mentioned above.

The C++ header file that defines our persistent classes and normally contains one or more ODB pragmas is compiled by both the ODB compiler to generate the database support code and the C++ compiler to build the application. Some C++ compilers issue warnings about pragmas that they do not recognize. There are several ways to deal with this problem which are covered at the end of this chapter in Section 14.9, "C++ Compiler Warnings".

14.1 Object Type Pragmas

A pragma with the `object` qualifier declares a C++ class as a persistent object type. The qualifier can be optionally followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`table`	table name for a persistent class	14.1.1
`pointer`	pointer type for a persistent class	14.1.2
`abstract`	persistent class is abstract	14.1.3
`readonly`	persistent class is read-only	14.1.4
`optimistic`	persistent class with the optimistic concurrency model	14.1.5
`no_id`	persistent class has no object id	14.1.6
`callback`	database operations callback	14.1.7
`schema`	database schema for a persistent class	14.1.8
`polymorphic`	persistent class is polymorphic	14.1.9
`session`	enable/disable session support for a persistent class	14.1.10
`definition`	definition location for a persistent class	14.1.11
`transient`	all non-virtual data members in a persistent class are transient	14.1.12
`sectionable`	support addition of new sections in derived classes	14.1.13
`deleted`	persistent class is soft-deleted	14.1.14
`bulk`	enable bulk operations for a persistent class	14.1.15
`options`	database options for a persistent class	14.1.16

14.1.1 `table`

The `table` specifier specifies the table name that should be used to store objects of the persistent class in a relational database. For example:

```
#pragma db object table("people")
class person
{
  ...
};
```

If the table name is not specified, the class name is used as the table name. The table name can be qualified with a database schema, for example:

```
#pragma db object table("census.people")
class person
{
  ...
};
```

For more information on database schemas and the format of the qualified names, refer to Section 14.1.8, "`schema`".

14.1.2 `pointer`

The `pointer` specifier specifies the object pointer type for the persistent class. The object pointer type is used to return, pass, and cache dynamically allocated instances of a persistent class. For example:

```
#pragma db object pointer(std::shared_ptr<person>)
class person
{
  ...
};
```

There are several ways to specify an object pointer with the `pointer` specifier. We can use a complete pointer type as shown in the example above. Alternatively, we can specify only the template name of a smart pointer in which case the ODB compiler will automatically append the class name as a template argument. The following example is therefore equivalent to the one above:

```
#pragma db object pointer(std::shared_ptr)
class person
{
  ...
};
```

If you would like to use the raw pointer as an object pointer, you can use `*` as a shortcut:

```
#pragma db object pointer(*) // Same as pointer(person*)
class person
{
  ...
};
```

If a pointer type is not explicitly specified, the default pointer, specified at the namespace level (Section 14.5.1, "`pointer`") or with the `--default-pointer` ODB compiler option, is used. If neither of these two mechanisms is used to specify the pointer, then the raw pointer is used by default.

For a more detailed discussion of object pointers, refer to Section 3.3, "Object and View Pointers".

14.1.3 `abstract`

The `abstract` specifier specifies that the persistent class is abstract. An instance of an abstract class cannot be stored in the database and is normally used as a base for other persistent classes. For example:

```
#pragma db object abstract
class person
{
  ...
};

#pragma db object
class employee: public person
{
  ...
};

#pragma db object
class contractor: public person
{
  ...
};
```

Persistent classes with pure virtual functions are automatically treated as abstract by the ODB compiler. For a more detailed discussion of persistent class inheritance, refer to Chapter 8, "Inheritance".

14.1.4 `readonly`

The `readonly` specifier specifies that the persistent class is read-only. The database state of read-only objects cannot be updated. In particular, this means that you cannot call the `database::update()` function (Section 3.10, "Updating Persistent Objects") for such objects. For example:

```
#pragma db object readonly
class person
{
  ...
};
```

Read-only and read-write objects can derive from each other without any restrictions. When a read-only object derives from a read-write object, the resulting whole object is read-only, including the part corresponding to the read-write base. On the other hand, when a read-write object derives from a read-only object, all the data members that correspond to the read-only base are treated as read-only while the rest is treated as read-write.

Note that it is also possible to declare individual data members (Section 14.4.12, "`readonly`") as well as composite value types (Section 14.3.6, "`readonly`") as read-only.

14.1.5 `optimistic`

The `optimistic` specifier specifies that the persistent class has the optimistic concurrency model. A class with the optimistic concurrency model must also specify the data member that is used to store the object version using the `version` pragma (Section 14.4.16, "`version`"). For example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version
  unsigned long long version_;
};
```

If a base class has the optimistic concurrency model, then all its derived classes will automatically have the optimistic concurrency model. The current implementation also requires that in any given inheritance hierarchy the object id and the version data members reside in the same class.

For a more detailed discussion of optimistic concurrency, refer to Chapter 12, "Optimistic Concurrency".

14.1.6 `no_id`

The `no_id` specifier specifies that the persistent class has no object id. For example:

```
#pragma db object no_id
class person
{
  ...
};
```

A persistent class without an object id has limited functionality. Such a class cannot be loaded with the `database::load()` or `database::find()` functions (Section 3.9, "Loading Persistent Objects"), updated with the `database::update()` function (Section 3.10, "Updating Persistent Objects"), or deleted with the `database::erase()` function (Section 3.11, "Deleting Persistent Objects"). To load and delete objects without ids you can use the `database::query()` (Chapter 4, "Querying the Database") and `database::erase_query()` (Section 3.11, "Deleting Persistent Objects") functions, respectively. There is no way to update such objects except by using native SQL statements (Section 3.12, "Executing Native SQL Statements").

Furthermore, persistent classes without object ids cannot have container data members nor can they be used in object relationships. Such objects are not entered into the session object cache (Section 11.1, "Object Cache") either.

To declare a persistent class with an object id, use the data member `id` specifier (Section 14.4.1, "`id`").

14.1.7 `callback`

The `callback` specifier specifies the persist class member function that should be called before and after a database operation is performed on an object of this class. For example:

```
#include <odb/callback.hxx>

#pragma db object callback(init)
class person
{
  ...

  void
  init (odb::callback_event, odb::database&);
};
```

The callback function has the following signature and can be overloaded for constant objects:

```
void
name (odb::callback_event, odb::database&);

void
name (odb::callback_event, odb::database&) const;
```

The first argument to the callback function is the event that triggered this call. The `odb::callback_event` enum-like type is defined in the `<odb/callback.hxx>` header file and has the following interface:

```
namespace odb
{
  struct callback_event
  {
    enum value
    {
      pre_persist,
      post_persist,
      pre_load,
      post_load,
      pre_update,
      post_update,
      pre_erase,
      post_erase
    };

    callback_event (value v);
    operator value () const;
  };
}
```

The second argument to the callback function is the database on which the operation is about to be performed or has just been performed. A callback function can be inline or virtual.

The callback function for the `*_persist`, `*_update`, and `*_erase` events is always called on the constant object reference while for the `*_load` events — always on the unrestricted reference.

If only the non-`const` version of the callback function is provided, then only the `*_load` events will be delivered. If only the `const` version is provided, then all the events will be delivered to this function. Finally, if both versions are provided, then the `*_load` events will be delivered to the non-`const` version while all others — to the `const` version. If you need to modify the object in one of the "`const`" events, then you can safely cast away `const`-ness using the `const_cast` operator if you know that none of the objects will be created const. Alternatively, if you cannot make this assumption, then you can declare the data members you wish to modify as `mutable`.

A database operations callback can be used to implement object-specific pre and post initializations, registrations, and cleanups. As an example, the following code fragment outlines an implementation of a `person` class that maintains the transient `age` data member in addition to the persistent date of birth. A callback is used to calculate the value of the former from the latter every time a `person` object is loaded from the database.

```
#include <odb/core.hxx>
#include <odb/callback.hxx>

#pragma db object callback(init)
class person
{
  ...

private:
  friend class odb::access;

  date born_;

  #pragma db transient
  unsigned short age_;

  void
  init (odb::callback_event e, odb::database&)
  {
    switch (e)
    {
    case odb::callback_event::post_load:
    {
      // Calculate age from the date of birth.
      ...
      break;
    }
    default:
      break;
    }
  }
};
```

14.1.8 `schema`

The `schema` specifier specifies a database schema that should be used for the persistent class.

In relational databases the term schema can refer to two related but ultimately different concepts. Normally it means a collection of tables, indexes, sequences, etc., that are created in the database or the actual DDL statements that create these database objects. Some database implementations support what would be more accurately called a *database namespace* but is also called a schema. In this sense, a schema is a separate namespace in which tables, indexes, sequences, etc., can be

created. For example, two tables that have the same name can coexist in the same database if they belong to different schemas. In this section when we talk about a schema, we refer to the *database namespace* meaning of this term.

When schemas are in use, a database object name is qualified with a schema. For example:

```
CREATE TABLE accounting.employee (...)

SELECT ... FROM accounting.employee WHERE ...
```

In the above example `accounting` is the schema and the `employee` table belongs to this schema.

Not all database implementations support schemas. Some implementation that don't support schemas (for example, MySQL, SQLite) allow the use of the above syntax to specify the database name. Yet others may support several levels of qualification. For example, Microsoft SQL Server has three levels starting with the linked database server, followed by the database, and then followed by the schema: `server1.company1.accounting.employee`. While the actual meaning of the qualifier in a qualified name vary from one database implementation to another, here we refer to all of them collectively as a schema.

In ODB, a schema for a table of a persistent class can be specified at the class level, C++ namespace level, or the file level. To assign a schema to a specific persistent class we can use the `schema` specifier, for example:

```
#pragma db object schema("accounting")
class employee
{
  ...
};
```

If we are also assigning a table name, then we can use a shorter notation by specifying both the schema and the table name in the `table` specifier:

```
#pragma db object table("accounting.employee")
class employee
{
  ...
};
```

If we want to assign a schema to all the persistent classes in a C++ namespace, then, instead of specifying the schema for each class, we can specify it once at the C++ namespace level. For example:

```
#pragma db namespace schema("accounting")
namespace accounting
{
  #pragma db object
  class employee
  {
    ...
  };

  #pragma db object
  class employer
  {
    ...
  };
}
```

If we want to assign a schema to all the persistent classes in a file, then we can use the `--schema` ODB compiler option. For example:

```
odb ... --schema accounting ...
```

An alternative to this approach with the same effect is to assign a schema to the global namespace:

```
#pragma db namespace() schema("accounting")
```

By default schema qualifications are accumulated starting from the persistent class, continuing with the namespace hierarchy to which this class belongs, and finishing with the schema specified with the `--schema` option. For example:

```
#pragma db namespace schema("audit_db")
namespace audit
{
  #pragma db namespace schema("accounting")
  namespace accounting
  {
    #pragma db object
    class employee
    {
      ...
    };
  }
}
```

If we compile the above code fragment with the `--schema server1` option, then the `employee` table will have the `server1.audit_db.accounting.employee` qualified name.

In some situations we may want to prevent such accumulation of the qualifications. To accomplish this we can use the so-called fully-qualified names, which have the empty leading name component. This is analogous to the C++ fully-qualified names in the `::accounting::employee` form. For example:

```
#pragma db namespace schema("accounting")
namespace accounting
{
  #pragma db object schema(".hr")
  class employee
  {
    ...
  };

  #pragma db object
  class employer
  {
    ...
  };
}
```

In the above code fragment, the `employee` table will have the `hr.employee` qualified name while the `employer` — `accounting.employer`. Note also that the empty leading name component is a special ODB syntax and is not propagated to the actual database names (using a name like `.hr.employee` to refer to a table will most likely result in an error).

Auxiliary database objects for a persistent class, such as indexes, sequences, triggers, etc., are all created in the same schema as the class table. By default, this is also true for the container tables. However, if you need to store a container table in a different schema, then you can provide a qualified name using the `table` specifier, for example:

```
#pragma db object table("accounting.employee")
class employee
{
  ...

  #pragma db object table("operations.projects")
  std::vector<std::string> projects_;
};
```

The standard syntax for qualified names used in the `schema` and `table` specifiers as well as the view `column` specifier (Section 14.4.10, "`column` (view)") has the "*name*`.`*name*..." form where, as discussed above, the leading name component can be empty to denote a fully qualified name. This form, however, doesn't work if one of the name components contains periods. To support such cases the alternative form is available: "*name*"`."`*name*"... For example:

```
#pragma db object table("accounting_1.2"."employee")
class employee
{
  ...
};
```

Finally, to specify an unqualified name that contains periods we can use the following special syntax:

```
#pragma db object schema(."accounting_1.2") table("employee")
class employee
{
  ...
};
```

Table prefixes (Section 14.5.2, "`table`") can be used as an alternative to database schemas if the target database system does not support schemas.

14.1.9 `polymorphic`

The `polymorphic` specifier specifies that the persistent class is polymorphic. For more information on polymorphism support, refer to Chapter 8, "Inheritance".

14.1.10 `session`

The `session` specifier specifies whether to enable session support for the persistent class. For example:

```
#pragma db object session        // Enable.
class person
{
  ...
};

#pragma db object session(true)  // Enable.
class employee
{
  ...
};

#pragma db object session(false) // Disable.
class employer
{
  ...
};
```

Session support is disabled by default unless the `--generate-session` ODB compiler option is specified or session support is enabled at the namespace level (Section 14.5.4, "`session`"). For more information on sessions, refer to Chapter 11, "Session".

14.1.11 `definition`

The `definition` specifier specifies an alternative *definition location* for the persistent class. By default, the ODB compiler generates the database support code for a persistent class when we compile the header file that defines this class. However, if the `definition` specifier is used, then the ODB compiler will instead generate the database support code when we compile the header file containing this pragma.

For more information on this functionality, refer to Section 14.3.7, "`definition`".

14.1.12 `transient`

The `transient` specifier instructs the ODB compiler to treat all non-virtual data members in the persistent class as transient (Section 14.4.1, "`transient`"). This specifier is primarily useful when declaring virtual data members, as discussed in Section 14.4.13, "`virtual`".

14.1.13 `sectionable`

The `sectionable` specifier instructs the ODB compiler to generate support for the addition of new object sections in derived classes in a hierarchy with the optimistic concurrency model. For more information on this functionality, refer to Section 9.2, "Sections and Optimistic Concurrency".

14.1.14 `deleted`

The `deleted` specifier marks the persistent class as soft-deleted. The single required argument to this specifier is the deletion version. For more information on this functionality, refer to Section 13.4, "Soft Object Model Changes".

14.1.15 `bulk`

The `bulk` specifier enables bulk operation support for the persistent class. The single required argument to this specifier is the batch size. For more information on this functionality, refer to Section 15.3, "Bulk Database Operations".

14.1.16 `options`

The `options` specifier specifies additional table definition options that should be used for the persistent class. For example:

```
#pragma db object options("PARTITION BY RANGE (age)")
class person
{
  ...

  unsigned short age_;
};
```

Table definition options for a container table can be specified with the `options` data member specifier (Section 14.4.8, "`options`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db options("PARTITION BY RANGE (index)")
  std::vector<std::string> aliases_;
};
```

14.2 View Type Pragmas

A pragma with the `view` qualifier declares a C++ class as a view type. The qualifier can be optionally followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`object`	object associated with a view	14.2.1
`table`	table associated with a view	14.2.2
`query`	view query condition	14.2.3
`pointer`	pointer type for a view	14.2.4
`callback`	database operations callback	14.2.5
`definition`	definition location for a view	14.2.6
`transient`	all non-virtual data members in a view are transient	14.2.7

For more information on view types refer to Chapter 10, "Views".

14.2.1 `object`

The `object` specifier specifies a persistent class that should be associated with the view. For more information on object associations refer to Section 10.1, "Object Views".

14.2.2 `table`

The `table` specifier specifies a database table that should be associated with the view. For more information on table associations refer to Section 10.3, "Table Views".

14.2.3 `query`

The `query` specifier specifies a query condition and, optionally, result modifiers for an object or table view or a native SQL query for a native view. An empty `query` specifier indicates that a native SQL query is provided at runtime. For more information on query conditions refer to Section 10.5, "View Query Conditions". For more information on native SQL queries, refer to Section 10.6, "Native Views".

14.2.4 `pointer`

The `pointer` specifier specifies the view pointer type for the view class. Similar to objects, the view pointer type is used to return dynamically allocated instances of a view class. The semantics of the `pointer` specifier for a view are the same as those of the `pointer` specifier for an object (Section 14.1.2, "`pointer`").

14.2.5 `callback`

The `callback` specifier specifies the view class member function that should be called before and after an instance of this view class is created as part of the query result iteration. The semantics of the `callback` specifier for a view are similar to those of the `callback` specifier for an object (Section 14.1.7, "`callback`") except that the only events that can trigger a callback call in the case of a view are `pre_load` and `post_load`.

14.2.6 `definition`

The `definition` specifier specifies an alternative *definition location* for the view class. By default, the ODB compiler generates the database support code for a view class when we compile the header file that defines this class. However, if the `definition` specifier is used, then the ODB compiler will instead generate the database support code when we compile the header file containing this pragma.

For more information on this functionality, refer to Section 14.3.7, "`definition`".

14.2.7 `transient`

The `transient` specifier instructs the ODB compiler to treat all non-virtual data members in the view class as transient (Section 14.4.1, "`transient`"). This specifier is primarily useful when declaring virtual data members, as discussed in Section 14.4.13, "`virtual`".

14.3 Value Type Pragmas

A pragma with the `value` qualifier describes a value type. It can be optionally followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`type`	database type for a value type	14.3.1
`id_type`	database type for a value type when used as an object id	14.3.2
`null/not_null`	type can/cannot be `NULL`	14.3.3
`default`	default value for a value type	14.3.4
`options`	database options for a value type	14.3.5
`readonly`	composite value type is read-only	14.3.6
`definition`	definition location for a composite value type	14.3.7
`transient`	all non-virtual data members in a composite value are transient	14.3.8
`unordered`	ordered container should be stored unordered	14.3.9
`index_type`	database type for a container's index type	14.3.10
`key_type`	database type for a container's key type	14.3.11
`value_type`	database type for a container's value type	14.3.12
`value_null/value_not_null`	container's value can/cannot be `NULL`	14.3.13
`id_options`	database options for a container's id column	14.3.14
`index_options`	database options for a container's index column	14.3.15
`key_options`	database options for a container's key column	14.3.16
`value_options`	database options for a container's value column	14.3.17
`id_column`	column name for a container's object id	14.3.18
`index_column`	column name for a container's index	14.3.19
`key_column`	column name for a container's key	14.3.20
`value_column`	column name for a container's value	14.3.21

Many of the value type specifiers have corresponding member type specifiers with the same names (Section 14.4, "Data Member Pragmas"). The behavior of such specifiers for members is similar to that for value types. The only difference is the scope. A particular value type specifier applies to all the members of this value type that don't have a pre-member version of the specifier, while the member specifier always applies only to a single member. Also, with a few excep-

tions, member specifiers take precedence over and override parameters specified with value specifiers.

14.3.1 type

The `type` specifier specifies the native database type that should be used for data members of this type. For example:

```
#pragma db value(bool) type("INT")

#pragma db object
class person
{
  ...

  bool married_; // Mapped to INT NOT NULL database type.
};
```

The ODB compiler provides the default mapping between common C++ types, such as `bool`, `int`, and `std::string` and the database types for each supported database system. For more information on the default mapping, refer to Part II, "Database Systems". The `null` and `not_null` (Section 14.3.3, "`null`/`not_null`") specifiers can be used to control the `NULL` semantics of a type.

In the above example we changed the mapping for the `bool` type which is now mapped to the `INT` database type. In this case, the `value` pragma is all that is necessary since the ODB compiler will be able to figure out how to store a boolean value as an integer in the database. However, there could be situations where the ODB compiler will not know how to handle the conversion between the C++ and database representations of a value. Consider, as an example, a situation where the boolean value is stored in the database as a string:

```
#pragma db value(bool) type("VARCHAR(5)")
```

The possible database values for the C++ `true` value could be `"true"`, or `"TRUE"`, or `"True"`. Or, maybe, all of the above could be valid. The ODB compiler has no way of knowing how your application wants to convert `bool` to a string and back. To support such custom value type mappings, ODB allows you to provide your own database conversion functions by specializing the `value_traits` class template. The `mapping` example in the `odb-examples` package shows how to do this for all the supported database systems.

14.3.2 id_type

The `id_type` specifier specifies the native database type that should be used for data members of this type that are designated as object identifiers (Section 14.4.1, "`id`"). In combination with the `type` specifier (Section 14.3.1, "`type`") `id_type` allows you to map a C++ type differently

depending on whether it is used in an ordinary member or an object id. For example:

```
#pragma db value(std::string) type("TEXT") id_type("VARCHAR(64)")

#pragma db object
class person
{
  ...

  #pragma db id
  std::string email_; // Mapped to VARCHAR(64) NOT NULL.

  std::string name_;  // Mapped to TEXT NOT NULL.
};
```

Note that there is no corresponding member type specifier for `id_type` since the desired result can be achieved with just the `type` specifier, for example:

```
#pragma db object
class person
{
  ...

  #pragma db id type("VARCHAR(128)")
  std::string email_;
};
```

14.3.3 `null`/`not_null`

The `null` and `not_null` specifiers specify that a value type or object pointer can or cannot be `NULL`, respectively. By default, value types are assumed not to allow `NULL` values while object pointers are assumed to allow `NULL` values. Data members of types that allow `NULL` values are mapped in a relational database to columns that allow `NULL` values. For example:

```
using string_ptr = std::shared_ptr<std::string>;
#pragma db value(string_ptr) type("TEXT") null

#pragma db object
class person
{
  ...

  string_ptr name_; // Mapped to TEXT NULL.
};

using person_ptr = std::shared_ptr<person>;
#pragma db value(person_ptr) not_null
```

The NULL semantics can also be specified on the per-member basis (Section 14.4.6, "null/not_null"). If both a type and a member have null/not_null specifiers, then the member specifier takes precedence. If a member specifier relaxes the NULL semantics (that is, if a member has the null specifier and the type has the explicit not_null specifier), then a warning is issued.

It is also possible to override a previously specified null/not_null specifier. This is primarily useful if a third-party type, for example, one provided by a profile library (Part III, "Profiles"), allows NULL values but in your object model data members of this type should never be NULL. In this case you can use the not_null specifier to disable NULL values for this type for the entire translation unit. For example:

```
// By default, null_string allows NULL values.
//
#include <null-string.hxx>

// Disable NULL values for all the null_string data members.
//
#pragma db value(null_string) not_null
```

For a more detailed discussion of the NULL semantics for values, refer to Section 7.3, "Pointers and NULL Value Semantics". For a more detailed discussion of the NULL semantics for object pointers, refer to Chapter 6, "Relationships".

14.3.4 default

The default specifier specifies the database default value that should be used for data members of this type. For example:

```
#pragma db value(std::string) default("")

#pragma db object
class person
{
  ...

  std::string name_; // Mapped to TEXT NOT NULL DEFAULT ''.
};
```

The semantics of the default specifier for a value type are similar to those of the default specifier for a data member (Section 14.4.7, "default").

14.3.5 `options`

The `options` specifier specifies additional column definition options that should be used for data members of this type. For example:

```
#pragma db value(std::string) options("COLLATE binary")

#pragma db object
class person
{
  ...

  std::string name_; // Mapped to TEXT NOT NULL COLLATE binary.
};
```

The semantics of the `options` specifier for a value type are similar to those of the `options` specifier for a data member (Section 14.4.8, "`options`").

14.3.6 `readonly`

The `readonly` specifier specifies that the composite value type is read-only. Changes to data members of a read-only composite value type are ignored when updating the database state of an object (Section 3.10, "Updating Persistent Objects") containing such a value type. Note that this specifier is only valid for composite value types. For example:

```
#pragma db value readonly
class person_name
{
  ...
};
```

Read-only and read-write composite values can derive from each other without any restrictions. When a read-only value derives from a read-write value, the resulting whole value is read-only, including the part corresponding to the read-write base. On the other hand, when a read-write value derives from a read-only value, all the data members that correspond to the read-only base are treated as read-only while the rest is treated as read-write.

Note that it is also possible to declare individual data members (Section 14.4.12, "`readonly`") as well as whole objects (Section 14.1.4, "`readonly`") as read-only.

14.3.7 `definition`

The `definition` specifier specifies an alternative *definition location* for the composite value type. By default, the ODB compiler generates the database support code for a composite value type when we compile the header file that defines this value type. However, if the `definition` specifier is used, then the ODB compiler will instead generate the database support code when we

compile the header file containing this pragma.

This mechanism is primarily useful for converting third-party types to ODB composite value types. In such cases we normally cannot modify the header files to add the necessary pragmas. It is also often inconvenient to compile these header files with the ODB compiler. With the `definition` specifier we can create a *wrapper header* that contains the necessary pragmas and instructs the ODB compiler to generate the database support code for a third-party type when we compile the wrapper header. As an example, consider `struct timeval` that is defined in the `<sys/time.h>` system header. This type has the following (or similar) definition:

```
struct timeval
{
  long tv_sec;
  long tv_usec;
};
```

If we would like to make this type an ODB composite value type, then we can create a wrapper header, for example `time-mapping.hxx`, with the following content:

```
#ifndef TIME_MAPPING_HXX
#define TIME_MAPPING_HXX

#include <sys/time.h>

#pragma db value(timeval) definition
#pragma db member(timeval::tv_sec) column("sec")
#pragma db member(timeval::tv_usec) column("usec")

#endif // TIME_MAPPING_HXX
```

If we now compile this header with the ODB compiler, the resulting `time-mapping-odb.?xx` files will contain the database support code for `struct timeval`. To use `timeval` in our persistent classes, we simply include the `time-mapping.hxx` header:

```
#include <sys/time.h>
#include "time-mapping.hxx"

#pragma db object
class object
{
  timeval timestamp;
};
```

14.3.8 `transient`

The `transient` specifier instructs the ODB compiler to treat all non-virtual data members in the composite value type as transient (Section 14.4.1, "`transient`"). This specifier is primarily useful when declaring virtual data members, as discussed in Section 14.4.13, "`virtual`".

14.3.9 `unordered`

The `unordered` specifier specifies that the ordered container should be stored unordered in the database. The database table for such a container will not contain the index column and the order in which elements are retrieved from the database may not be the same as the order in which they were stored. For example:

```
using names = std::vector<std::string>;
#pragma db value(names) unordered
```

For a more detailed discussion of ordered containers and their storage in the database, refer to Section 5.1, "Ordered Containers".

14.3.10 `index_type`

The `index_type` specifier specifies the native database type that should be used for the ordered container's index column. The semantics of `index_type` are similar to those of the `type` specifier (Section 14.3.1, "`type`"). The native database type is expected to be an integer type. For example:

```
using names = std::vector<std::string>;
#pragma db value(names) index_type("SMALLINT UNSIGNED")
```

14.3.11 `key_type`

The `key_type` specifier specifies the native database type that should be used for the map container's key column. The semantics of `key_type` are similar to those of the `type` specifier (Section 14.3.1, "`type`"). For example:

```
using age_weight_map = std::map<unsigned short, float>;
#pragma db value(age_weight_map) key_type("INT UNSIGNED")
```

14.3.12 `value_type`

The `value_type` specifier specifies the native database type that should be used for the container's value column. The semantics of `value_type` are similar to those of the `type` specifier (Section 14.3.1, "`type`"). For example:

```
using names = std::vector<std::string>;
#pragma db value(names) value_type("VARCHAR(255)")
```

The `value_null` and `value_not_null` (Section 14.3.13, "`value_null`/`value_not_null`") specifiers can be used to control the `NULL` semantics of a value column.

14.3.13 `value_null`/`value_not_null`

The `value_null` and `value_not_null` specifiers specify that the container type's element value can or cannot be `NULL`, respectively. The semantics of `value_null` and `value_not_null` are similar to those of the `null` and `not_null` specifiers (Section 14.3.3, "`null`/`not_null`"). For example:

```
#pragma db object
class account
{
  ...
};

using accounts = std::vector<std::shared_ptr<account>>;
#pragma db value(accounts) value_not_null
```

For set and multiset containers (Section 5.2, "Set and Multiset Containers") the element value is automatically treated as not allowing a `NULL` value.

14.3.14 `id_options`

The `id_options` specifier specifies additional column definition options that should be used for the container's id column. For example:

```
using nicknames = std::vector<std::string>;
#pragma db value(nicknames) id_options("COLLATE binary")
```

The semantics of the `id_options` specifier for a container type are similar to those of the `id_options` specifier for a container data member (Section 14.4.29, "`id_options`").

14.3.15 `index_options`

The `index_options` specifier specifies additional column definition options that should be used for the container's index column. For example:

```
using nicknames = std::vector<std::string>;
#pragma db value(nicknames) index_options("ZEROFILL")
```

The semantics of the `index_options` specifier for a container type are similar to those of the `index_options` specifier for a container data member (Section 14.4.30, "`index_options`").

14.3.16 `key_options`

The `key_options` specifier specifies additional column definition options that should be used for the container's key column. For example:

```
using properties = std::map<std::string, std::string>;
#pragma db value(properties) key_options("COLLATE binary")
```

The semantics of the `key_options` specifier for a container type are similar to those of the `key_options` specifier for a container data member (Section 14.4.31, "`key_options`").

14.3.17 `value_options`

The `value_options` specifier specifies additional column definition options that should be used for the container's value column. For example:

```
using nicknames = std::set<std::string>;
#pragma db value(nicknames) value_options("COLLATE binary")
```

The semantics of the `value_options` specifier for a container type are similar to those of the `value_options` specifier for a container data member (Section 14.4.32, "`value_options`").

14.3.18 `id_column`

The `id_column` specifier specifies the column name that should be used to store the object id in the container's table. For example:

```
using names = std::vector<std::string>;
#pragma db value(names) id_column("id")
```

If the column name is not specified, then `object_id` is used by default.

14.3.19 `index_column`

The `index_column` specifier specifies the column name that should be used to store the element index in the ordered container's table. For example:

```
using names = std::vector<std::string>;
#pragma db value(names) index_column("name_number")
```

If the column name is not specified, then `index` is used by default.

14.3.20 `key_column`

The `key_column` specifier specifies the column name that should be used to store the key in the map container's table. For example:

```
using age_weight_map = std::map<unsigned short, float>;
#pragma db value(age_weight_map) key_column("age")
```

If the column name is not specified, then `key` is used by default.

14.3.21 `value_column`

The `value_column` specifier specifies the column name that should be used to store the element value in the container's table. For example:

```
using age_weight_map = std::map<unsigned short, float>;
#pragma db value(age_weight_map) value_column("weight")
```

If the column name is not specified, then `value` is used by default.

14.4 Data Member Pragmas

A pragma with the `member` qualifier or a positioned pragma without a qualifier describes a data member. It can be optionally followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`id`	member is an object id	14.4.1
`auto`	id is assigned by the database	14.4.2
`type`	database type for a member	14.4.3
`id_type`	database type for a member when used as an object id	14.4.4
`get/set/access`	member accessor/modifier expressions	14.4.5
`null/not_null`	member can/cannot be `NULL`	14.4.6
`default`	default value for a member	14.4.7
`options`	database options for a member	14.4.8

`column`	column name for a member of an object or composite value	14.4.9
`column`	column name for a member of a view	14.4.10
`transient`	member is not stored in the database	14.4.11
`readonly`	member is read-only	14.4.12
`virtual`	declare a virtual data member	14.4.13
`inverse`	member is an inverse side of a bidirectional relationship	14.4.14
`on_delete`	`ON DELETE` clause for object pointer member	14.4.15
`version`	member stores object version	14.4.16
`index`	define database index for a member	14.4.17
`unique`	define unique database index for a member	14.4.18
`unordered`	ordered container should be stored unordered	14.4.19
`table`	table name for a container	14.4.20
`load/update`	loading/updating behavior for a section	14.4.21
`section`	member belongs to a section	14.4.22
`added`	member is soft-added	14.4.23
`deleted`	member is soft-deleted	14.4.24
`index_type`	database type for a container's index type	14.4.25
`key_type`	database type for a container's key type	14.4.26
`value_type`	database type for a container's value type	14.4.27
`value_null/value_not_null`	container's value can/cannot be `NULL`	14.4.28
`id_options`	database options for a container's id column	14.4.29
`index_options`	database options for a container's index column	14.4.30
`key_options`	database options for a container's key column	14.4.31
`value_options`	database options for a container's value column	14.4.32
`id_column`	column name for a container's object id	14.4.33

`index_column`	column name for a container's index	14.4.34
`key_column`	column name for a container's key	14.4.35
`value_column`	column name for a container's value	14.4.36
`points_to`	establish relationship without object pointer	14.4.37

Many of the member specifiers have corresponding value type specifiers with the same names (Section 14.3, "Value Type Pragmas"). The behavior of such specifiers for members is similar to that for value types. The only difference is the scope. A particular value type specifier applies to all the members of this value type that don't have a pre-member version of the specifier, while the member specifier always applies only to a single member. Also, with a few exceptions, member specifiers take precedence over and override parameters specified with value specifiers.

14.4.1 `id`

The `id` specifier specifies that the data member contains the object id. In a relational database, an identifier member is mapped to a primary key. For example:

```
#pragma db object
class person
{
  ...

  #pragma db id
  std::string email_;
};
```

Normally, every persistent class has a data member designated as an object's identifier. However, it is possible to declare a persistent class without an id using the object `no_id` specifier (Section 14.1.6, "`no_id`").

Note also that the `id` specifier cannot be used for data members of composite value types or views.

14.4.2 `auto`

The `auto` specifier specifies that the object's identifier is automatically assigned by the database. Only a member that was designated as an object id can have this specifier. For example:

```
#pragma db object
class person
{
  ...

  #pragma db id auto
  unsigned long long id_;
};
```

Note that automatically-assigned object ids are not reused. If you have a high object turnover (that is, objects are routinely made persistent and then erased), then care must be taken not to run out of object ids. In such situations, using a 64-bit integer as the identifier type is a safe choice.

For additional information on the automatic identifier assignment, refer to Section 3.8, "Making Objects Persistent".

Note also that the `auto` specifier cannot be specified for data members of composite value types or views.

14.4.3 `type`

The `type` specifier specifies the native database type that should be used for the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db type("INT")
  bool married_;
};
```

The `null` and `not_null` (Section 14.4.6, "`null`/`not_null`") specifiers can be used to control the `NULL` semantics of a data member. It is also possible to specify the database type on the per-type instead of the per-member basis using the value `type` specifier (Section 14.3.1, "`type`").

14.4.4 `id_type`

The `id_type` specifier specifies the native database type that should be used for the data member when it is part of an object identifier. This specifier only makes sense when applied to a member of a composite value type that is used for both id and non-id members. For example:

```
#pragma db value
class name
{
  ...

  #pragma db type("VARCHAR(256)") id_type("VARCHAR(64)")
  std::string first_;

  #pragma db type("VARCHAR(256)") id_type("VARCHAR(64)")
  std::string last_;
};

#pragma db object
class person
{
  ...

  #pragma db id
  name name_;  // name_.first_, name_.last_ mapped to VARCHAR(64)

  name alias_; // alias_.first_, alias_.last_ mapped to VARCHAR(256)
};
```

14.4.5 get/set/access

The `get` and `set` specifiers specify the data member accessor and modifier expressions, respectively. If provided, the generated database support code will use these expressions to access and modify the data member when performing database operations. The `access` specifier can be used as a shortcut to specify both the accessor and modifier if they happen to be the same.

In its simplest form the accessor or modifier expression can be just a name. Such a name should resolve either to another data member of the same type or to a suitable accessor or modifier member function. For example:

```
#pragma db object
class person
{
  ...

public:
  const std::string& name () const;
  void name (const std::string&);
private:
  #pragma db access(name)
  std::string name_;
};
```

A suitable accessor function is a `const` member function that takes no arguments and whose return value can be implicitly converted to the `const` reference to the member type (`const std::string&` in the example above). An accessor function that returns a `const` reference to the data member is called *by-reference accessor*. Otherwise, it is called *by-value accessor*.

A suitable modifier function can be of two forms. It can be the so called *by-reference modifier* which is a member function that takes no arguments and returns a non-`const` reference to the data member (`std::string&` in the example above). Alternatively, it can be the so called *by-value modifier* which is a member function taking a single argument — the new value — that can be implicitly initialized from a variable of the member type (`std::string` in the example above). The return value of a by-value modifier, if any, is ignored. If both by-reference and by-value modifiers are available, then ODB prefers the by-reference version since it is more efficient. For example:

```
#pragma db object
class person
{
  ...

public:
  std::string get_name () const;      // By-value accessor.
  std::string& set_name ();           // By-reference modifier.
  void set_name (std::string const&); // By-value modifier.
private:
  #pragma db get(get_name) \ // Uses by-value accessor.
             set(set_name)   // Uses by-reference modifier.
  std::string name_;
};
```

Note that in many cases it is not necessary to specify accessor and modifier functions explicitly since the ODB compiler will try to discover them automatically in case the data member will be inaccessible to the generated code. In particular, in both of the above examples the ODB compiler would have successfully discovered the necessary functions. For more information on this functionality, refer to Section 3.2, "Declaring Persistent Objects and Values".

Note also that by-value accessors and by-value modifiers cannot be used for certain data members in certain situations. These limitations are discussed in more detail later in this section.

Accessor and modifier expressions can be more elaborate than simple names. An accessor expression is any C++ expression that can be used to initialize a `const` reference to the member type. Similar to accessor functions, which are just a special case of accessor expressions, an accessor expression that evaluates to a `const` reference to the data member is called *by-reference accessor expression*. Otherwise, it is called *by-value accessor expression*.

Modifier expressions can also be of two forms: *by-reference modifier expression* and *by-value modifier expression* (again, modifier functions are just a special case of modifier expressions). A by-reference modifier expression is any C++ expression that evaluates to the non-`const` reference to the member type. A by-value modifier expression can be a single or multiple (separated by semicolon) C++ statements with the effect of setting the new member value.

There are three special placeholders that are recognized by the ODB compiler in accessor and modifier expressions. The first is the `this` keyword which denotes a reference (note: not a pointer) to the persistent object. In accessor expressions this reference is `const` while in modifier expressions it is non-`const`. If an expression does not contain the `this` placeholder, then the ODB compiler automatically prefixes it with the `this.` sequence.

The second placeholder, the `(?)` sequence, is used to denote the new value in by-value modifier expressions. The ODB compiler replaces the question mark with the variable name, keeping the surrounding parenthesis.

The third placeholder, the `(!)` sequence, is used to denote the database instance in the modifier expressions. The ODB compiler replaces the exclamation mark with the reference to the database, keeping the surrounding parenthesis. The database instance can, for example, be used to load an object pointer.

The following example shows a few more interesting accessor and modifier expressions:

```
#pragma db value
struct point
{
  point (int, int);

  int x;
  int y;
};

#pragma db object
class person
{
  ...

  public:
    const char* name () const;
    void name (const char*);
  private:
    #pragma db get(std::string (this.name ())) \
               set(name ((?).c_str ())) // The same as this.name (...).
    std::string name_;

  public:
    const std::unique_ptr<account>& acc () const;
    void acc (std::unique_ptr<account>);
```

```
  private:
    #pragma db set(acc (std::move (?)))
    std::unique_ptr<account> acc_;

  public:
    int loc_x () const
    int loc_y () const
    void loc_x (int);
    void loc_y (int);
  private:
    #pragma db get(point (this.loc_x (), this.loc_y ()))    \
               set(this.loc_x ((?).x); this.loc_y ((?).y))
    point loc_;
};
```

When the data member is of an array type, then the terms "reference" and "member type" in the above discussion should be replaced with "pointer" and "array element type", respectively. That is, the accessor expression for an array member is any C++ expression that can be used to initialize a `const` pointer to the array element type, and so on. The following example shows common accessor and modifier signatures for array members:

```
#pragma db object
class person
{
  ...

  public:
    const char* id () const; // By-reference accessor.
    void id (const char*);   // By-value modifier.
  private:
    char id_[16];

  public:
    const char* pub_key () const; // By-reference accessor.
    char* pub_key ();             // By-reference modifier.
  private:
    char pub_key_[2048];
};
```

Accessor and modifier expressions can be used with data members of simple value, composite value, container, and object pointer types. They can be used for data members in persistent classes, composite value types, and views. There is also a mechanism related to accessors and modifiers called virtual data members and which is discussed in Section 14.4.13, "`virtual`".

There are, however, certain limitations when it comes to using by-value accessor and modifier expressions. First of all, if a by-value modifier is used, then the data member type should be default-constructible. Furthermore, a composite value type that has a container member cannot be modified with a by-value modifier. Only a by-reference modifier expression can be used. The

ODB compiler will detect such cases and issue diagnostics. For example:

```
#pragma db value
struct name
{
  std::string first_;
  std::string last_;
  std::vector<std::string> aliases_;
};

#pragma db object
class person
{
  ...

public:
  const name& name () const;
  void name (const name&);
private:
  #pragma db access(name) // Error: by-value modifier.
  name name_;
};
```

In certain database systems it is also not possible to use by-value accessor and modifier expression with certain database types. The ODB compiler is only able to detect such cases and issue diagnostics if you specified accessor/modifier function names as opposed to custom expressions. For more information on these database and type-specific limitations, refer to the "Limitations" sections in Part II, "Database Systems".

14.4.6 `null`/`not_null`

The `null` and `not_null` specifiers specify that the data member can or cannot be `NULL`, respectively. By default, data members of basic value types for which database mapping is provided by the ODB compiler do not allow `NULL` values while data members of object pointers allow `NULL` values. Other value types, such as those provided by the profile libraries (Part III, "Profiles"), may or may not allow `NULL` values, depending on the semantics of each value type. Consult the relevant documentation to find out more about the `NULL` semantics for such value types. A data member containing the object id (Section 14.4.1, "`id`") is automatically treated as not allowing a `NULL` value. Data members that allow `NULL` values are mapped in a relational database to columns that allow `NULL` values. For example:

```
#pragma db object
class person
{
  ...

  #pragma db null
  std::string name_;
```

```
};

#pragma db object
class account
{
  ...

  #pragma db not_null
  std::shared_ptr<person> holder_;
};
```

The `NULL` semantics can also be specified on the per-type basis (Section 14.3.3, "`null`/`not_null`"). If both a type and a member have `null`/`not_null` specifiers, then the member specifier takes precedence. If a member specifier relaxes the `NULL` semantics (that is, if a member has the `null` specifier and the type has the explicit `not_null` specifier), then a warning is issued.

For a more detailed discussion of the `NULL` semantics for values, refer to Section 7.3, "Pointers and `NULL` Value Semantics". For a more detailed discussion of the `NULL` semantics for object pointers, refer to Chapter 6, "Relationships".

14.4.7 `default`

The `default` specifier specifies the database default value that should be used for the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db default(-1)
  int age_;              // Mapped to INT NOT NULL DEFAULT -1.
};
```

A default value can be the special `null` keyword, a `bool` literal (`true` or `false`), an integer literal, a floating point literal, a string literal, or an enumerator name. If you need to specify a default value that is an expression, for example an SQL function call, then you can use the `options` specifier (Section 14.4.8, "`options`") instead. For example:

```
enum gender {male, female, undisclosed};

#pragma db object
class person
{
  ...

  #pragma db default(null)
```

```
  odb::nullable<std::string> middle_; // DEFAULT NULL

  #pragma db default(false)
  bool married_;                      // DEFAULT 0/FALSE

  #pragma db default(0.0)
  float weight_;                      // DEFAULT 0.0

  #pragma db default("Mr")
  string title_;                      // DEFAULT 'Mr'

  #pragma db default(undisclosed)
  gender gender_;                     // DEFAULT 2/'undisclosed'

  #pragma db options("DEFAULT CURRENT_TIMESTAMP()")
  date timestamp_;                    // DEFAULT CURRENT_TIMESTAMP()
};
```

Default values specified as enumerators are only supported for members that are mapped to an `ENUM` or an integer type in the database, which is the case for the automatic mapping of C++ enums and enum classes to suitable database types as performed by the ODB compiler. If you have mapped a C++ enum or enum class to another database type, then you should use a literal corresponding to that type to specify the default value. For example:

```
enum gender {male, female, undisclosed};
#pragma db value(gender) type("VARCHAR(11)")

#pragma db object
class person
{
  ...

  #pragma db default("undisclosed")
  gender gender_;                   // DEFAULT 'undisclosed'
};
```

A default value can also be specified on the per-type basis (Section 14.3.4, "`default`"). An empty `default` specifier can be used to reset a default value that was previously specified on the per-type basis. For example:

```
#pragma db value(std::string) default("")

#pragma db object
class person
{
  ...

  #pragma db default()
  std::string name_;   // No default value.
};
```

A data member containing the object id (Section 14.4.1, "id") is automatically treated as not having a default value even if its type specifies a default value.

Note also that default values do not affect the generated C++ code in any way. In particular, no automatic initialization of data members with their default values is performed at any point. If you need such an initialization, you will need to implement it yourself, for example, in your persistent class constructors. The default values only affect the generated database schemas and, in the context of ODB, are primarily useful for schema evolution.

Additionally, the `default` specifier cannot be specified for view data members.

14.4.8 `options`

The `options` specifier specifies additional column definition options that should be used for the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db options("CHECK(email != '')")
  std::string email_; // Mapped to TEXT NOT NULL CHECK(email != '').
};
```

Note that if specified for the container member, then instead of the column definition options it specifies the table definition options for the container table (Section 14.1.16, "options").

Options can also be specified on the per-type basis (Section 14.3.5, "options"). By default, options are accumulating. That is, the ODB compiler first adds all the options specified for a value type followed by all the options specified for a data member. To clear the accumulated options at any point in this sequence you can use an empty `options` specifier. For example:

```
#pragma db value(std::string) options("COLLATE binary")

#pragma db object
class person
{
  ...

  std::string first_; // TEXT NOT NULL COLLATE binary

  #pragma db options("CHECK(last != '')")
  std::string last_;  // TEXT NOT NULL COLLATE binary CHECK(last != '')

  #pragma db options()
  std::string title_; // TEXT NOT NULL
```

```
  #pragma db options() options("CHECK(email != '')")
  std::string email_; // TEXT NOT NULL CHECK(email != '')
};
```

ODB provides dedicated specifiers for specifying column types (Section 14.4.3, "`type`"), `NULL` constraints (Section 14.4.6, "`null`/`not_null`"), and default values (Section 14.4.7, "`default`"). For ODB to function correctly these specifiers should always be used instead of the opaque `options` specifier for these components of a column definition.

Note also that the `options` specifier cannot be specified for view data members.

14.4.9 `column` (object, composite value)

The `column` specifier specifies the column name that should be used to store the data member of a persistent class or composite value type in a relational database. For example:

```
#pragma db object
class person
{
  ...

  #pragma db id column("person_id")
  unsigned long long id_;
};
```

For a member of a composite value type, the `column` specifier specifies the column name prefix. Refer to Section 7.2.2, "Composite Value Column and Table Names" for details.

If the column name is not specified, it is derived from the member's so-called public name. A public member name is obtained by removing the common data member name decorations, such as leading and trailing underscores, the `m_` prefix, etc.

14.4.10 `column` (view)

The `column` specifier can be used to specify the associated object data member, the potentially qualified column name, or the column expression for the data member of a view class. For more information, refer to Section 10.1, "Object Views" and Section 10.3, "Table Views".

14.4.11 `transient`

The `transient` specifier instructs the ODB compiler not to store the data member in the database. For example:

```
#pragma db object
class person
{
  ...

  date born_;

  #pragma db transient
  unsigned short age_; // Computed from born_.
};
```

This pragma is usually used on computed members, pointers and references that are only meaningful in the application's memory, as well as utility members such as mutexes, etc.

14.4.12 `readonly`

The `readonly` specifier specifies that the data member of an object or composite value type is read-only. Changes to a read-only data member are ignored when updating the database state of an object (Section 3.10, "Updating Persistent Objects") containing such a member. Since views are read-only, it is not necessary to use this specifier for view data members. Object id (Section 14.4.1, "`id`") and inverse (Section 14.4.14, "`inverse`") data members are automatically treated as read-only and must not be explicitly declared as such. For example:

```
#pragma db object
class person
{
  ...

  #pragma db readonly
  date born_;
};
```

Besides simple value members, object pointer, container, and composite value members can also be declared read-only. A change of a pointed-to object is ignored when updating the state of a read-only object pointer. Similarly, any changes to the number or order of elements or to the element values themselves are ignored when updating the state of a read-only container. Finally, any changes to the members of a read-only composite value type are also ignored when updating the state of such a composite value.

ODB automatically treats `const` data members as read-only. For example, the following `person` object is equivalent to the above declaration for the database persistence purposes:

```
#pragma db object
class person
{
  ...

  const date born_; // Automatically read-only.
};
```

When declaring an object pointer `const`, make sure to declare the pointer as `const` rather than (or in addition to) the object itself. For example:

```
#pragma db object
class person
{
  ...

  const person* father_; // Read-write pointer to a read-only object.
  person* const mother_; // Read-only pointer to a read-write object.
};
```

Note that in case of a wrapper type (Section 7.3, "Pointers and `NULL` Value Semantics"), both the wrapper and the wrapped type must be `const` in order for the ODB compiler to automatically treat the data member as read-only. For example:

```
#pragma db object
class person
{
  ...

  const std::unique_ptr<const date> born_;
};
```

Read-only members are useful when dealing with asynchronous changes to the state of a data member in the database which should not be overwritten. In other cases, where the state of a data member never changes, declaring such a member read-only allows ODB to perform more efficient object updates. In such cases, however, it is conceptually more correct to declare such a data member as `const` rather than as read-only.

Note that it is also possible to declare composite value types (Section 14.3.6, "`readonly`") as well as whole objects (Section 14.1.4, "`readonly`") as read-only.

14.4.13 `virtual`

The `virtual` specifier is used to declare a virtual data member in an object, view, or composite value type. A virtual data member is an *imaginary* data member that is only used for the purpose of database persistence. A virtual data member does not actually exist (that is, occupy space) in the C++ class. Note also that virtual data members have nothing to do with C++ virtual functions

or virtual inheritance. Specifically, no virtual function call overhead is incurred when using virtual data members.

To declare a virtual data member we must specify the data member name using the `member` specifier. We must also specify the data member type with the `virtual` specifier. Finally, the virtual data member declaration must also specify the accessor and modifier expressions, unless suitable accessor and modifier functions can automatically be found by the ODB compiler (Section 14.4.5, "`get/set/access`"). For example:

```
#pragma db object
class person
{
  ...

  // Transient real data member that actually stores the data.
  //
  #pragma db transient
  std::string name_;

  // Virtual data member.
  //
  #pragma db member(name) virtual(std::string) access(name_)
};
```

From the pragma language point of view, a virtual data member behaves exactly like a normal data member. Specifically, we can reference the virtual data member after it has been declared and use positioned pragmas before its declaration. For example:

```
#pragma db object
class person
{
  ...

  #pragma db transient
  std::string name_;

  #pragma db access(name_)
  #pragma db member(name) virtual(std::string)
};

#pragma db member(person::name) column("person_name")
#pragma db index member(person::name)
```

We can also declare a virtual data member outside the class scope:

```
#pragma db object
class person
{
  ...

  std::string name_;
};

#pragma db member(person::name_) transient
#pragma db member(person::name) virtual(std::string) access(name_)
```

The order of data members determines the order of columns in the resulting table. The order of virtual data members in relation to other data members (virtual or not) is the order of declaration, with virtual data members declared outside of the class coming last. This order, however, can be adjusted with the `before` and `after` specifiers. One of these specifiers without a parameter places the virtual member at the beginning or at the end of the members list, respectively. Alternatively, we can specify the member (virtual or not) before/after which this virtual member should be placed. For example:

```
#pragma db object
class person
{
  ...

  #pragma db id auto
  unsigned long long id_;

  #pragma db member(first) virtual(std::string) before
  #pragma db member(last)  virtual(std::string) after(first)
};
```

The order of columns in the resulting table will be: `first`, `last`, `id`.

While in the above examples using virtual data members doesn't seem to yield any benefits, this mechanism can be useful in a number of situations. As one example, consider the need to aggregate or dis-aggregate a data member:

```
#pragma db object
class person
{
  ...

  #pragma db transient
  std::pair<std::string, std::string> name_;

  #pragma db member(first) virtual(std::string) access(name_.first)
  #pragma db member(last) virtual(std::string) access(name_.second)
};
```

We can also use virtual data members to implement composite object ids that are spread over multiple data members:

```
#pragma db value
struct name
{
  name () {}
  name (std::string const& f, std::string const& l)
    : first (f), last(l) {}

  std::string first;
  std::string last;
};

#pragma db object
class person
{
  ...

  #pragma db transient
  std::string first_;

  #pragma db transient
  std::string last_;

  #pragma db member(name) virtual(name) id                        \
             get(::name (this.first_, this.last_))                \
             set(this.first_ = (?).first; this.last_ = (?).last)
};
```

Another common situation that calls for virtual data members is a class that uses the pimpl idiom. While the following code fragment outlines the idea, for details refer to the `pimpl` example in the `odb-examples` package.

```
#pragma db object
class person
{
public:
  std::string const& name () const;
  void name (std::string const&);

  unsigned short age () const;
  void age (unsigned short);

  ...

private:
  class impl;

  #pragma db transient
```

```
  impl* pimpl_;

  #pragma db member(name) virtual(std::string)   // Uses name().
  #pragma db member(age) virtual(unsigned short) // Uses age().
};
```

The above example also shows that names used for virtual data members (`name` and `age` in our case) can be the same as the names of accessor/modifier functions. The only names that virtual data members cannot clash with are those of other data members, virtual or real.

A common pattern in the above examples is the need to declare the real data member that actually stores the data as transient. If all the real data members in a class are treated as transient, then we can use the class-level `transient` specifier (Section 14.1.12, "`transient` (object)", Section 14.3.8, "`transient` (composite value)", Section 14.2.7, "`transient` (view)") instead of doing it for each individual member. For example:

```
#pragma db object transient
class person
{
  ...

  std::string first_; // Transient.
  std::string last_;  // Transient.

  #pragma db member(name) virtual(name) ...
};
```

The ability to treat all the real data members as transient becomes more important if we don't know the names of these data members. This is often the case when we are working with third-party types that document the accessor and modifier functions but not the names of their private data members. As an example, consider the `point` class defined in a third-party `<point>` header file:

```
class point
{
public:
  point ();
  point (int x, int y);

  int x () const;
  int y () const;

  void x (int);
  void y (int);

private:
  ...
};
```

To convert this class to an ODB composite value type we could create the `point-mapping.hxx` file with the following content:

```
#include <point>

#pragma db value(point) transient definition
#pragma db member(point::x) virtual(int)
#pragma db member(point::y) virtual(int)
```

Virtual data members can be of simple value, composite value, container, or object pointer types. They can be used in persistent classes, composite value types, and views.

14.4.14 `inverse`

The `inverse` specifier specifies that the data member of an object pointer or a container of object pointers type is an inverse side of a bidirectional object relationship. The single required argument to this specifier is the corresponding data member name in the referenced object. For example:

```
class person;

#pragma db object pointer(std::shared_ptr)
class employer
{
  ...

  std::vector<std::shared_ptr<person>> employees_;
};

#pragma db object pointer(std::shared_ptr)
class person
{
  ...

  #pragma db inverse(employee_)
  std::weak_ptr<employer> employer_;
};
```

An inverse member does not have a corresponding column or, in case of a container, table in the resulting database schema. Instead, the column or table from the referenced object is used to retrieve the relationship information. Only ordered and set containers can be used for inverse members. If an inverse member is of an ordered container type, it is automatically marked as unordered (Section 14.4.19, "`unordered`").

For a more detailed discussion of inverse members, refer to Section 6.2, "Bidirectional Relationships".

14.4.15 on_delete

The on_delete specifier specifies the on-delete semantics for a data member of an object pointer or a container of object pointers type. The single required argument to this specifier must be either cascade or set_null.

The on_delete specifier is translated directly to the corresponding ON DELETE SQL clause. That is, if cascade is specified, then when a pointed-to object is erased from the database, the database state of the pointing object is automatically erased as well. If set_null is specified, then when a pointed-to object is erased from the database, the database state of the pointing object is automatically updated to set the pointer column to NULL. For example:

```
#pragma db object
class employer
{
  ...

  #pragma db id auto
  unsigned long long id_;
};

#pragma db object
class person
{
  ...

  #pragma db on_delete(cascade)
  employer* employer_;
};

unsigned long long id;

{
  employer e;
  person p;
  p.employer_ = &e;

  transaction t (db.begin ());

  id = db.persist (e);
  db.persist (p);

  t.commit ();
}

{
  transaction t (db.begin ());

  // Database state of the person object is erased as well.
```

```
  //
  db.erase<employer> (id);

  t.commit ();
}
```

Note that this is a database-level functionality and care must be taken in order not to end up with inconsistent object states in the application's memory and database. The following example illustrates the kind of problems one may encounter:

```
#pragma db object
class employer
{
  ...
};

#pragma db object
class person
{
  ...

  #pragma db on_delete(set_null)
  employer* employer_;
};

employer e;
person p;
p.employer_ = &e;

{
  transaction t (db.begin ());
  db.persist (e);
  db.persist (p);
  t.commit ();
}

{
  transaction t (db.begin ());

  // The employer column is set to NULL in the database but
  // not the p.employer_ data member in the application.
  //
  db.erase (e);
  t.commit ();
}

{
  transaction t (db.begin ());

  // Override the employer column with an invalid pointer.
  //
```

```
  db.update (p);

  t.commit ();
}
```

Note that even optimistic concurrency will not resolve such issues unless you are using database-level support for optimistic concurrency as well (for example, `ROWVERSION` in SQL Server).

The `on_delete` specifier is only valid for non-inverse object pointer data members. If the `set_null` semantics is used, then the pointer must allow the `NULL` value.

14.4.16 `version`

The `version` specifier specifies that the data member stores the object version used to support optimistic concurrency. If a class has a version data member, then it must also be declared as having the optimistic concurrency model using the `optimistic` pragma (Section 14.1.5, "`optimistic`"). For example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version
  unsigned long long version_;
};
```

A version member must be of an integral C++ type and must map to an integer or similar database type. Note also that object versions are not reused. If you have a high update frequency, then care must be taken not to run out of versions. In such situations, using a 64-bit integer as the version type is a safe choice.

For a more detailed discussion of optimistic concurrency, refer to Chapter 12, "Optimistic Concurrency".

14.4.17 `index`

The `index` specifier instructs the ODB compiler to define a database index for the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db index
  std::string name_;
};
```

For more information on defining database indexes, refer to Section 14.7, "Index Definition Pragmas".

14.4.18 `unique`

The `index` specifier instructs the ODB compiler to define a unique database index for the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db unique
  std::string name_;
};
```

For more information on defining database indexes, refer to Section 14.7, "Index Definition Pragmas".

14.4.19 `unordered`

The `unordered` specifier specifies that the member of an ordered container type should be stored unordered in the database. The database table for such a member will not contain the index column and the order in which elements are retrieved from the database may not be the same as the order in which they were stored. For example:

```
#pragma db object
class person
{
  ...

  #pragma db unordered
  std::vector<std::string> nicknames_;
};
```

For a more detailed discussion of ordered containers and their storage in the database, refer to Section 5.1, "Ordered Containers".

14.4.20 `table`

The `table` specifier specifies the table name that should be used to store the contents of the container member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db table("nicknames")
  std::vector<std::string> nicknames_;
};
```

If the table name is not specified, then the container table name is constructed by concatenating the object's table name, underscore, and the public member name. The public member name is obtained by removing the common member name decorations, such as leading and trailing underscores, the `m_` prefix, etc. In the example above, without the `table` specifier, the container's table name would have been `person_nicknames`.

The `table` specifier can also be used for members of composite value types. In this case it specifies the table name prefix for container members inside the composite value type. Refer to Section 7.2.2, "Composite Value Column and Table Names" for details.

The container table name can be qualified with a database schema, for example:

```
#pragma db object
class person
{
  ...

  #pragma db table("extras.nicknames")
  std::vector<std::string> nicknames_;
};
```

For more information on database schemas and the format of the qualified names, refer to Section 14.1.8, "`schema`".

14.4.21 `load/update`

The `load` and `update` specifiers specify the loading and updating behavior for an object section, respectively. Valid values for the `load` specifier are `eager` (default) and `lazy`. Valid values for the `update` specifier are `always` (default), `change`, and `manual`. For more infor-

mation on object sections, refer to Chapter 9, "Sections".

14.4.22 `section`

The `section` specifier indicates that a data member of a persistent class belongs to an object section. The single required argument to this specifier is the name of the section data member. This specifier can only be used on direct data members of a persistent class. For more information on object sections, refer to Chapter 9, "Sections".

14.4.23 `added`

The `added` specifier marks the data member as soft-added. The single required argument to this specifier is the addition version. For more information on this functionality, refer to Section 13.4, "Soft Object Model Changes".

14.4.24 `deleted`

The `deleted` specifier marks the data member as soft-deleted. The single required argument to this specifier is the deletion version. For more information on this functionality, refer to Section 13.4, "Soft Object Model Changes".

14.4.25 `index_type`

The `index_type` specifier specifies the native database type that should be used for an ordered container's index column of the data member. The semantics of `index_type` are similar to those of the `type` specifier (Section 14.4.3, "`type`"). The native database type is expected to be an integer type. For example:

```
#pragma db object
class person
{
  ...

  #pragma db index_type("SMALLINT UNSIGNED")
  std::vector<std::string> nicknames_;
};
```

14.4.26 `key_type`

The `key_type` specifier specifies the native database type that should be used for a map container's key column of the data member. The semantics of `key_type` are similar to those of the `type` specifier (Section 14.4.3, "`type`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db key_type("INT UNSIGNED")
  std::map<unsigned short, float> age_weight_map_;
};
```

14.4.27 `value_type`

The `value_type` specifier specifies the native database type that should be used for a container's value column of the data member. The semantics of `value_type` are similar to those of the `type` specifier (Section 14.4.3, "`type`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db value_type("VARCHAR(255)")
  std::vector<std::string> nicknames_;
};
```

The `value_null` and `value_not_null` (Section 14.4.28, "`value_null`/`value_not_null`") specifiers can be used to control the `NULL` semantics of a value column.

14.4.28 `value_null`/`value_not_null`

The `value_null` and `value_not_null` specifiers specify that a container's element value for the data member can or cannot be `NULL`, respectively. The semantics of `value_null` and `value_not_null` are similar to those of the `null` and `not_null` specifiers (Section 14.4.6, "`null`/`not_null`"). For example:

```
#pragma db object
class person
{
  ...
};

#pragma db object
class account
{
  ...
```

```
  #pragma db value_not_null
  std::vector<std::shared_ptr<person>> holders_;
};
```

For set and multiset containers (Section 5.2, "Set and Multiset Containers") the element value is automatically treated as not allowing a `NULL` value.

14.4.29 `id_options`

The `id_options` specifier specifies additional column definition options that should be used for a container's id column of the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db id options("COLLATE binary")
  std::string name_;

  #pragma db id_options("COLLATE binary")
  std::vector<std::string> nicknames_;
};
```

The semantics of `id_options` are similar to those of the `options` specifier (Section 14.4.8, "`options`").

14.4.30 `index_options`

The `index_options` specifier specifies additional column definition options that should be used for a container's index column of the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db index_options("ZEROFILL")
  std::vector<std::string> nicknames_;
};
```

The semantics of `index_options` are similar to those of the `options` specifier (Section 14.4.8, "`options`").

14.4.31 key_options

The key_options specifier specifies additional column definition options that should be used for a container's key column of the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db key_options("COLLATE binary")
  std::map<std::string, std::string> properties_;
};
```

The semantics of key_options are similar to those of the options specifier (Section 14.4.8, "options").

14.4.32 value_options

The value_options specifier specifies additional column definition options that should be used for a container's value column of the data member. For example:

```
#pragma db object
class person
{
  ...

  #pragma db value_options("COLLATE binary")
  std::set<std::string> nicknames_;
};
```

The semantics of value_options are similar to those of the options specifier (Section 14.4.8, "options").

14.4.33 id_column

The id_column specifier specifies the column name that should be used to store the object id in a container's table for the data member. The semantics of id_column are similar to those of the column specifier (Section 14.4.9, "column"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db id_column("person_id")
  std::vector<std::string> nicknames_;
};
```

If the column name is not specified, then `object_id` is used by default.

14.4.34 `index_column`

The `index_column` specifier specifies the column name that should be used to store the element index in an ordered container's table for the data member. The semantics of `index_column` are similar to those of the `column` specifier (Section 14.4.9, "`column`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db index_column("nickname_number")
  std::vector<std::string> nicknames_;
};
```

If the column name is not specified, then `index` is used by default.

14.4.35 `key_column`

The `key_column` specifier specifies the column name that should be used to store the key in a map container's table for the data member. The semantics of `key_column` are similar to those of the `column` specifier (Section 14.4.9, "`column`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db key_column("age")
  std::map<unsigned short, float> age_weight_map_;
};
```

If the column name is not specified, then `key` is used by default.

14.4.36 `value_column`

The `value_column` specifier specifies the column name that should be used to store the element value in a container's table for the data member. The semantics of `value_column` are similar to those of the `column` specifier (Section 14.4.9, "`column`"). For example:

```
#pragma db object
class person
{
  ...

  #pragma db value_column("weight")
  std::map<unsigned short, float> age_weight_map_;
};
```

If the column name is not specified, then `value` is used by default.

14.4.37 `points_to`

The `points_to` specifier allows the establishment of object relationships without using object pointers. For example:

```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string id_;
};

#pragma db object
class person
{
  ...

  #pragma db points_to(employer)
  std::string employer_;
};
```

14.5 Namespace Pragmas

A pragma with the `namespace` qualifier describes a C++ namespace. Similar to other qualifiers, `namespace` can also refer to a named C++ namespace, for example:

```
namespace test
{
  ...
}

#pragma db namespace(test) ...
```

To refer to the global namespace in the `namespace` qualifier the following special syntax is used:

```
#pragma db namespace() ....
```

The `namespace` qualifier can be optionally followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`pointer`	pointer type for persistent classes and views inside a namespace	14.5.1
`table`	table name prefix for persistent classes inside a namespace	14.5.2
`schema`	database schema for persistent classes inside a namespace	14.5.3
`session`	enable/disable session support for persistent classes inside a namespace	14.5.4

14.5.1 `pointer`

The `pointer` specifier specifies the default pointer type for persistent classes and views inside the namespace. For example:

```
#pragma db namespace pointer(std::shared_ptr)
namespace accounting
{
  #pragma db object
  class employee
  {
    ...
  };

  #pragma db object
  class employer
  {
    ...
  };
}
```

There are only two valid ways to specify a pointer with the `pointer` specifier at the namespace level. We can specify the template name of a smart pointer in which case the ODB compiler will automatically append the class name as a template argument. Or we can use `*` to denote a raw pointer.

Note also that we can always override the default pointer specified at the namespace level for any persistent class or view inside this namespace. For example:

```
#pragma db namespace pointer(std::unique_ptr)
namespace accounting
{
  #pragma db object pointer(std::shared_ptr)
  class employee
  {
    ...
  };

  #pragma db object
  class employer
  {
    ...
  };
}
```

For a more detailed discussion of object and view pointers, refer to Section 3.3, "Object and View Pointers".

14.5.2 `table`

The `table` specifier specifies a table prefix that should be added to table names of persistent classes inside the namespace. For example:

```
#pragma db namespace table("acc_")
namespace accounting
{
  #pragma db object table("employees")
  class employee
  {
    ...
  };

  #pragma db object table("employers")
  class employer
  {
    ...
  };
}
```

In the above example the resulting table names will be `acc_employees` and `acc_employers`.

The table name prefix can also be specified with the `--table-prefix` ODB compiler option. Note that table prefixes specified at the namespace level as well as with the command line option are accumulated. For example:

```
#pragma db namespace() table("audit_")

#pragma db namespace table("hr_")
namespace hr
{
  #pragma db object table("employees")
  class employee
  {
    ...
  };
}

#pragma db object table("employers")
class employer
{
  ...
};
```

If we compile the above example with the `--table-prefix test_` option, then the `employee` class table will be called `test_audit_hr_employees` and `employer` — `test_audit_employers`.

Table prefixes can be used as an alternative to database schemas (Section 14.1.8, "`schema`") if the target database system does not support schemas.

14.5.3 `schema`

The `schema` specifier specifies a database schema that should be used for persistent classes inside the namespace. For more information on specifying a database schema refer to Section 14.1.8, "`schema`".

14.5.4 `session`

The `session` specifier specifies whether to enable session support for persistent classes inside the namespace. For example:

```
#pragma db namespace session
namespace hr
{
  #pragma db object                // Enabled.
  class employee
  {
    ...
  };

  #pragma db object session(false) // Disabled.
  class employer
```

```
  {
    ...
  };
}
```

Session support is disabled by default unless the `--generate-session` ODB compiler option is specified. Session support specified at the namespace level can be overridden on the per object basis (Section 14.1.10, "`session`"). For more information on sessions, refer to Chapter 11, "Session".

14.6 Object Model Pragmas

A pragma with the `model` qualifier describes the whole C++ object model. For example:

```
#pragma db model ...
```

The `model` qualifier can be followed, in any order, by one or more specifiers summarized in the table below:

Specifier	Summary	Section
`version`	object model version	14.6.1

14.6.1 `version`

The `version` specifier specifies the object model version when schema evolution support is used. The first two required arguments to this specifier are the base and current model versions, respectively. The third optional argument specifies whether the current version is open for changes. Valid values for this argument are `open` (the default) and `closed`. For more information on this functionality, refer to Chapter 13, "Database Schema Evolution".

14.7 Index Definition Pragmas

While it is possible to manually add indexes to the generated database schema, it is more convenient to do this as part of the persistent class definitions. A pragma with the `index` qualifier describes a database index. It has the following general format:

```
#pragma db index[("<name>")]                          \
          [unique|type("<type>")]                     \
          [method("<method>")]                        \
          [options("<index-options>")]                \
          member(<name>[, "<column-options>"])... \
          members(<name>[,<name>...])...
```

The `index` qualifier can optionally specify the index name. If the index name is not specified, then one is automatically derived by appending the `_i` suffix to the column name of the index member. If the name is not specified and the index contains multiple members, then the index definition is invalid.

The optional `type`, `method`, and `options` clauses specify the index type, for example `UNIQUE`, index method, for example `BTREE`, and index options, respectively. The `unique` clause is a shortcut for `type("UNIQUE")`. Note that not all database systems support specifying an index method or options. For more information on the database system-specific index types, methods, and options, refer to Part II, "Database Systems".

To specify index members we can use the `member` or `members` clauses, or a mix of the two. The `member` clause allows us to specify a single index member with optional column options, for example, `"ASC"`. If we need to create a composite index that contains multiple members, then we can repeat the `member` clause several times or, if the members don't have any column options, we can use a single `members` clause instead. Similar to the index type, method, and options, the format of column options is database system-specific. For more details, refer to Part II, "Database Systems".

The following code fragment shows some typical examples of index definitions:

```
#pragma db object
class object
{
  ...

  int x;
  int y;
  int z1;
  int z2;

  // An index for member x with automatically-assigned name x_i.
  //
  #pragma db index member(x)

  // A unique index named y_index for member y which is sorted in
  // the descending order. The index is using the BTREE method.
  //
  #pragma db index("y_index") unique method("BTREE") member(y, "DESC")

  // A composite BITMAP index named z_i for members z1 and z2.
  //
  #pragma db index("z_i") type("BITMAP") members(z1, z2)
};
```

ODB also offers a shortcut for defining an index with the default method and options for a single data member. Such an index can be defined using the `index` (Section 14.4.17, "`index`") or `unique` (Section 14.4.18, "`unique`") member specifier. For example:

```
#pragma db object
class object
{
  ...

  #pragma db index
  int x;

  #pragma db type("INT") unique
  int y;
};
```

The above example is semantically equivalent to the following more verbose version:

```
#pragma db object
class object
{
  ...

  int x;

  #pragma db type("INT")
  int y;

  #pragma db index member(x)
  #pragma db index unique member(y)
};
```

While it is convenient to define an index inside a persistent class, it is also possible to do that out of the class body. In this case, the index name, if specified, must be prefixed with the potentially-qualified class name. For example:

```
namespace n
{
  #pragma db object
  class object
  {
    ...

    int x;
    int y;
  };

  // An index for member x in persistent class object with automatically-
  // assigned name x_i.
  //
```

```
  #pragma db index(object) member(x)
}

// An index named y_index for member y in persistent class n::object.
//
#pragma db index(n::object::"y_index") member(y)
```

It is possible to define an index on a member that is of a composite value type or on some of its nested members. For example:

```
#pragma db value
struct point
{
  int x;
  int y;
  int z;
};

#pragma db object
class object
{
  // An index that includes all of the p1's nested members.
  //
  #pragma db index
  point p1;

  point p2;

  // An index that includes only p2.x and p2.y.
  //
  #pragma db index("p2_xy_i") members(p2.x, p2.y)
};
```

When generating a schema for a container member (Chapter 5, "Containers"), ODB automatically defines two indexes on the container table. One is for the object id that references the object table and the other is for the index column in case the container is ordered (Section 5.1, "Ordered Containers"). By default these indexes use the default index name, type, method, and options. The `index` pragma allows us to customize these two indexes by recognizing the special `id` and `index` nested member names when specified after a container member. For example:

```
#pragma db object
class object
{
  std::vector<int> v;

  // Change the container id index name.
  //
  #pragma db index("id_index") member(v.id)
```

```
  // Change the container index index method.
  //
  #pragma db index method("BTREE") member(v.index)
};
```

14.8 Database Type Mapping Pragmas

A pragma with the `map` qualifier describes a mapping between either two C++ types (Section 14.8.1, "C++ Type Mapping Pragmas") or two database types (Section 14.8.2, "Database Type Mapping Pragmas").

14.8.1 C++ Type Mapping Pragmas

A pragma with the `map` qualifier can describe a mapping between two C++ types. For each database system ODB provides built-in support for mapping a standard set of C++ types, such as integers, strings, binary, etc. However, many codebases may use custom versions of corresponding (or similar) C++ types. While it is possible to add support for such custom types by providing a suitable specialization of the `value_traits` class template, it is often simpler to define their mapping in terms of one of the already supported types by specifying the conversion between the two types. This mechanism can also be used to redefine the mapping for one of the standard types. For example, we could map `bool` to `std::string` in order to save boolean values as string literals. This mechanism can also be used with composite value types.

The `map` pragma for C++ types has the following format:

```
#pragma db map type(<name>)  \
               as(<name>)    \
               [to(<expr>)]  \
               [from(<expr>)]
```

The `type` clause specifies the name of the C++ type that we are mapping. We will refer to it as the *mapped type* from now on.

The `as` clause specifies the name of the C++ type that we are mapping the mapped type to. We will refer to it as the *interface type* from now on.

The optional `to` and `from` clauses specify the C++ conversion expressions between the mapped type and the interface type. The `to` expression converts from the interface type to the mapped type and `from` converts in the other direction. If no explicit conversion is required for either direction, then the corresponding clause can be omitted. If the conversion expressions are specified, then they must contain the special `(?)` placeholder which will be replaced with the actual value to be converted.

As an example, suppose we have the `path` type which represents a filesystem path and which we wish to store in the database. Suppose also that it has a constructor that allows implicit conversion of `std::string` to `path` as well as the `string()` member function which returns the `std::string` representation of `path`. This is how we can map `path` to `std::string` to be able to store it in the database:

```
#pragma db map type(path)        \
               as(std::string)   \
               from((?).string ())
```

Notice that we could omit the `to` expression because `std::string` can be implicitly converted to `path`. In this case the implied expression is equivalent to `to((?))`.

As a more elaborate example, the following `map` pragma maps the standard `chrono` `time_point` type (that is, a timestamp) as `uint64` with the stored values representing the number of nanoseconds since the UNIX epoch:

```
#pragma db map                                                        \
  type(std::chrono::system_clock::time_point)                         \
  as(std::uint64_t)                                                   \
  to(std::chrono::duration_cast<std::chrono::nanoseconds> (           \
       (?).time_since_epoch ()).count ())                             \
  from(std::chrono::system_clock::time_point (                        \
    std::chrono::duration_cast<std::chrono::system_clock::duration> ( \
      std::chrono::nanoseconds (?))))
```

And the following example shows how to redefine the mapping for one of the standard types:

```
#pragma db map type(bool)                 \
               as(std::string)            \
               to((?) ? "true" : "false") \
               from((?) == "true")
```

14.8.2 Database Type Mapping Pragmas

A pragma with the `map` qualifier can describe a mapping between two database types. For each database system ODB provides built-in support for a core set of database types, such as integers, strings, binary, etc. However, many database systems provide extended types such as geospatial types, user-defined types, and collections (arrays, table types, key-value stores, etc). In order to support such extended types, ODB allows us to map them to one of the built-in types, normally a string or a binary. Given the text or binary representation of the data we can then extract it into our chosen C++ data type and thus establish a mapping between an extended database type and its C++ equivalent.

The `map` pragma for database types has the following format:

```
#pragma db map type("<regex>")  \
               as("<subst>")    \
               [to("<subst>")]  \
               [from("<subst>")]
```

The `type` clause specifies the name of the database type that we are mapping. We will refer to it as the *mapped type* from now on. The name of the mapped type is a Perl-like regular expression pattern that is matched in the case-insensitive mode.

The `as` clause specifies the name of the database type that we are mapping the mapped type to. We will refer to it as the *interface type* from now on. The name of the interface type is a regular expression substitution and should expand to a name of a database type for which ODB provides built-in support.

The optional `to` and `from` clauses specify the database conversion expressions between the mapped type and the interface type. The `to` expression converts from the interface type to the mapped type and `from` converts in the other direction. If no explicit conversion is required for either direction, then the corresponding clause can be omitted. The conversion expressions are regular expression substitutions. They must contain the special `(?)` placeholder which will be replaced with the actual value to be converted. Turning on SQL statement tracing (Section 3.13, "Tracing SQL Statement Execution") can be useful for debugging conversion expressions. This allows you to see the substituted expressions as used in the actual statements.

As an example, the following `map` pragma maps the PostgreSQL array of `INTEGER`'s to `TEXT`:

```
#pragma db map type("INTEGER *\\[(\\d*)\\]") \
               as("TEXT")                    \
               to("(?)::INTEGER[$1]")        \
               from("(?)::TEXT")
```

With the above mapping we can now have a persistent class that has a member of the PostgreSQL array type:

```
#pragma db object
class object
{
  ...

  #pragma db type("INTEGER[]")
  std::string array_;
};
```

In PostgreSQL the array literal has the `{n1,n2,...}` form. As a result, we need to make sure that we pass the correct text representation in the `array_` member, for example:

```
object o;
o.array_ = "{1,2,3}";
db.persist (o);
```

Of course, `std::string` is not the most natural representation of an array of integers in C++. Instead, `std::vector<int>` would have been much more appropriate. To add support for mapping `std::vector<int>` to PostgreSQL `INTEGER[]` we need to provide a `value_traits` specialization that implements conversion between the PostgreSQL text representation of an array and `std::vector<int>`. Below is a sample implementation:

```
namespace odb
{
  namespace pgsql
  {
    template <>
    class value_traits<std::vector<int>, id_string>
    {
    public:
      using value_type = std::vector<int>;
      using query_type = value_type;
      using image_type = details::buffer;

      static void
      set_value (value_type& v,
                 const details::buffer& b,
                 std::size_t n,
                 bool is_null)
      {
        v.clear ();

        if (!is_null)
        {
          char c;
          std::istringstream is (std::string (b.data (), n));

          is >> c; // '{'

          for (c = static_cast<char> (is.peek ()); c != '}'; is >> c)
          {
            v.push_back (int ());
            is >> v.back ();
          }
        }
      }

      static void
      set_image (details::buffer& b,
```

```
                std::size_t& n,
                bool& is_null,
                const value_type& v)
      {
        is_null = false;
        std::ostringstream os;

        os << '{';

        for (value_type::const_iterator i (v.begin ()), e (v.end ());
             i != e;)
        {
          os << *i;

          if (++i != e)
            os << ',';
        }

        os << '}';

        const std::string& s (os.str ());
        n = s.size ();

        if (n > b.capacity ())
          b.capacity (n);

        std::memcpy (b.data (), s.c_str (), n);
      }
    };
  }
}
```

Once this specialization is included in the generated code (see the `mapping` example in the `odb-examples` package for details), we can use `std::vector<int>` instead of `std::string` in our persistent class:

```
#pragma db object
class object
{
  ...

  #pragma db type("INTEGER[]")
  std::vector<int> array_;
};
```

If we wanted to always map `std::vector<int>` to PostgreSQL `INTEGER[]`, then we could instead write:

```
using int_vector = std::vector<int>;
#pragma db value(int_vector) type("INTEGER[]")

#pragma db object
class object
{
  ...

  std::vector<int> array_; // Mapped to INTEGER[].
};
```

While the above example only shows how to handle PostgreSQL arrays, other types in PostgreSQL and in other databases can be supported in a similar way. The `odb-tests` package contains a set of tests in the `<database>/custom` directories that, for each database, shows how to provide custom mapping for some of the extended types.

14.9 C++ Compiler Warnings

When a C++ header file defining persistent classes and containing ODB pragmas is used to build the application, the C++ compiler may issue warnings about pragmas that it doesn't recognize. There are several ways to deal with this problem. The easiest is to disable such warnings using one of the compiler-specific command line options or warning control pragmas. This method is described in the following sub-section for popular C++ compilers.

There are also several C++ compiler-independent methods that we can employ. The first is to use the `PRAGMA_DB` macro, defined in `<odb/core.hxx>`, instead of using `#pragma db` directly. This macro expands to the ODB pragma when compiled with the ODB compiler and to an empty declaration when compiled with other compilers. The following example shows how we can use this macro:

```
#include <odb/core.hxx>

PRAGMA_DB(object)
class person
{
  ...

  PRAGMA_DB(id)
  unsigned long long id_;
};
```

An alternative to using the `PRAGMA_DB` macro is to group the `#pragma db` directives in blocks that are conditionally included into compilation only when compiled with the ODB compiler. For example:

```
class person
{
  ...

  unsigned long long id_;
};

#ifdef ODB_COMPILER
#  pragma db object(person)
#  pragma db member(person::id_) id
#endif
```

The disadvantage of this approach is that it can quickly become overly verbose when positioned pragmas are used.

14.9.1 GNU C++

GNU g++ does not issue warnings about unknown pragmas unless requested with the `-Wall` command line option. To disable only the unknown pragma warning, we can add the `-Wno-unknown-pragmas` option after `-Wall`, for example:

```
g++ -Wall -Wno-unknown-pragmas ...
```

14.9.2 Visual C++

Microsoft Visual C++ issues an unknown pragma warning (C4068) at warning level 1 or higher. This means that unless we have disabled the warnings altogether (level 0), we will see this warning.

To disable this warning via the compiler command line, we can add the `/wd4068` C++ compiler option in Visual Studio 2008 and earlier. In Visual Studio 2010 and later there is now a special GUI field where we can enter warning numbers that should be disabled. Simply enter 4068 into this field.

We can also disable this warning for only a specific header or a fragment of a header using the warning control pragma. For example:

```
#include <odb/core.hxx>

#pragma warning (push)
#pragma warning (disable:4068)

#pragma db object
class person
{
  ...
```

```
  #pragma db id
  unsigned long long id_;
};

#pragma warning (pop)
```

14.9.3 Sun C++

The Sun C++ compiler does not issue warnings about unknown pragmas unless the `+w` or `+w2` option is specified. To disable only the unknown pragma warning we can add the `-erroff=unknownpragma` option anywhere on the command line, for example:

```
CC +w -erroff=unknownpragma ...
```

14.9.4 IBM XL C++

IBM XL C++ issues an unknown pragma warning (1540-1401) by default. To disable this warning we can add the `-qsuppress=1540-1401` command line option, for example:

```
xlC -qsuppress=1540-1401 ...
```

14.9.5 HP aC++

HP aC++ (aCC) issues an unknown pragma warning (2161) by default. To disable this warning we can add the `+W2161` command line option, for example:

```
aCC +W2161 ...
```

14.9.6 Clang

Clang does not issue warnings about unknown pragmas unless requested with the `-Wall` command line option. To disable only the unknown pragma warning, we can add the `-Wno-unknown-pragmas` option after `-Wall`, for example:

```
clang++ -Wall -Wno-unknown-pragmas ...
```

We can also disable this warning for only a specific header or a fragment of a header using the warning control pragma. For example:

```
#include <odb/core.hxx>

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunknown-pragmas"

#pragma db object
class person
{
```

```
  ...

  #pragma db id
  unsigned long long id_;
};

#pragma clang diagnostic pop
```

15 Advanced Techniques and Mechanisms

This chapter covers more advanced techniques and mechanisms provided by ODB that may be useful in certain situations.

15.1 Transaction Callbacks

The ODB transaction class (`odb::transaction`) allows an application to register a callback that will be called after the transaction is finalized, that is, committed or rolled back. This mechanism can be used, for example, to restore values that were updated during the transaction execution to their original states if the transaction is rolled back.

The callback management interface of the `transaction` class is shown below.

```
namespace odb
{
  class transaction
  {
    ...

  public:
    static const unsigned short event_commit = 0x01;
    static const unsigned short event_rollback = 0x02;
    static const unsigned short event_all = event_commit | event_rollback;

    using callback_type = void (*) (
      unsigned short event, void* key, unsigned long long data);

    void
    callback_register (callback_type callback,
                       void* key,
                       unsigned short event = event_all,
                       unsigned long long data = 0,
                       transaction** state = 0);


    void
    callback_unregister (void* key);

    void
    callback_update (void* key,
                     unsigned short event,
                     unsigned long long data = 0,
                     transaction** state = 0);
  }
}
```

The `callback_register()` function registers a post-commit/rollback callback. The `callback` argument is the function that should be called. The `key` argument is used by the transaction to identify this callback. It is also normally used to pass an address of the data object on which the callback function will work. The `event` argument is the bitwise-or of the events that should trigger the callback.

The optional data argument can be used to store any POD user data that doesn't exceed 8 bytes in size and doesn't require alignment greater than `unsigned long long`. For example, we could store an old value of a flag or a counter that needs to be restored in case of a roll back.

The optional `state` argument can be used to indicate that the callback has been unregistered because the transaction was finalized. In this case the transaction automatically resets the passed pointer to 0. This is primarily useful if we are interested in only one of the events (commit or rollback).

The `callback_unregister()` function unregisters a previously registered callback. If the number of registered callbacks is large, then this can be a slow operation. Generally, the callback mechanism is optimized for cases where the callbacks stay registered until the transaction is finalized.

Note also that you don't need to unregister a callback that has been called or auto-reset using the `state` argument passed to `callback_register()`. This function does nothing if the key is not found.

The `callback_update()` function can be used to update the `event`, `data`, and `state` values of a previously registered callback. Similar to `callback_unregister()`, this is a potentially slow operation.

When the callback is called, it is passed the event that triggered it, as well as the `key` and `data` values that were passed to the `callback_register()` function. Note also that the order in which the callbacks are called is unspecified. The rollback event can be triggered by an exception. In this case, if the callback throws, the program will be terminated.

The following example shows how we can use transaction callbacks together with database operation callbacks (Section 14.1.7, "`callback`") to manage the object's "dirty" flag.

```
#include <odb/callback.hxx>
#include <odb/transaction.hxx>

#pragma db object callback(update)
class object
{
  ...

  #pragma db transient
  mutable bool dirty_;
```

```
  // Non-NULL value indicates that we are registered
  // with this transaction.
  //
  #pragma db transient
  mutable odb::transaction* tran_;

  void
  update (odb::callback_event e, odb::database&) const
  {
    using namespace odb::core;

    if (e == callback_event::post_update)
      return;

    // Mark the object as clean again but register a
    // transaction callback in case the update is rolled
    // back.
    //
    tran_ = &transaction::current ();
    tran_->callback_register (&rollback,
                              const_cast<object*> (this),
                              transaction::event_rollback,
                              0,
                              &tran_);
    dirty_ = false;
  }

  static void
  rollback (unsigned short, void* key, unsigned long long)
  {
    // Restore the dirty flag since the changes have been
    // rolled back.
    //
    object& o (*static_cast<object*> (key));
    o.dirty_ = true;
  }

  ~object ()
  {
    // Unregister the callback if we are going away before
    // the transaction.
    //
    if (tran_ != 0)
      tran_->callback_unregister (this);
  }
};
```

15.2 Persistent Class Template Instantiations

Similar to composite value types (Section 7.2, "Composite Value Types"), a persistent object can be defined as an instantiation of a C++ class template, for example:

```
template <typename T>
class person
{
  ...

  T first_;
  T last_;
};

using std_person = person<std::string>;

#pragma db object(std_person)
#pragma db member(std_person::last_) id
```

Note that the database support code for such a persistent object is generated when compiling the header containing the `db object` pragma and not the header containing the template definition or the `using` alias. This allows us to use templates defined in other files, for example:

```
#include <utility> // std::pair

using person = std::pair<unsigned int, std::string>;
#pragma db object(person)
#pragma db member(person::first) id auto column("id")
#pragma db member(person::second) column("name")
```

You may also have to explicitly specify the object type in calls to certain `database` class functions due to the inability do distinguish, at the API level, between smart pointers and persistent objects defined as class template instantiations. For example:

```
person p;

db.update (p); // Error.
db.reload (p); // Error.
db.erase (p);  // Error.

db.update<person> (p); // Ok.
db.reload<person> (p); // Ok.
db.erase<person> (p);  // Ok.
```

It also makes sense to factor persistent data members that do not depend on template arguments into a common, non-template base class. The following more realistic example illustrates this approach:

```
#pragma db object abstract
class base_common
{
  ...

  #pragma db id auto
  unsigned long long id;
};

template <typename T>
class base: public base_common
{
  ...

  T value;
};

using string_base = base<std::string>;
#pragma db object(string_base) abstract

#pragma db object
class derived: public string_base
{
  ...
};
```

15.3 Bulk Database Operations

Some database systems supported by ODB provide a mechanism, often called bulk or batch statement execution, that allows us to execute the same SQL statement on multiple sets of data at once and with a single database API call (or equivalent). This often results in significantly better performance if we need to execute the same statement for a large number of data sets (thousands to millions).

ODB translates this mechanism to bulk operations which allow us to persist, update, or erase a range of objects in the database. Currently, from all the database systems supported by ODB, only Oracle, Microsoft SQL Server, and PostgreSQL are capable of bulk operations (but see Section 19.5.7, "Bulk Operations Support" for PostgreSQL limitations). There is also currently no emulation of the bulk API for other databases nor dynamic multi-database support. As a result, if you are using dynamic multi-database support, you will need to "drop down" to static support in order to access the bulk API. Refer to Chapter 16, "Multi-Database Support" for details.

As we will discuss later in this section, bulk operations have complex failure semantics that is dictated by the underlying database API. As a result, support for bulk persist, update, and erase is limited to persistent classes for which these operations can be performed with a single database statement execution. In particular, bulk operations are not available for polymorphic objects (Section 8.2, "Polymorphism Inheritance") or objects that have containers (inverse containers of

object pointers are an exception). Furthermore, for objects that have sections (Chapter 9, "Sections") the bulk update operation will only be available if all the sections are manually-updated. On the other hand, bulk operations are supported for objects that use optimistic concurrency (Chapter 12, "Optimistic Concurrency") or have no object id (Section 14.1.6, "no_id").

To enable the generation of bulk operation support for a persistent class we use the `bulk` pragma. For example:

```
#pragma db object bulk(5000)
class person
{
  ...

  #pragma db id auto
  unsigned long long id;
};
```

The single argument to the `bulk` pragma is the batch size. The batch size specifies the maximum number of data sets that should be handled with a single underlying statement execution (or equivalent). If the range that we want to perform the bulk operation on contains more objects than the batch size, then ODB will split this operation into multiple underlying statement executions (batches). To illustrate this point with an example, suppose we want to persist 53,000 objects and the batch size is 5,000. ODB will then execute the statement 11 times, the first 10 times with 5,000 data sets each, and the last time with the remaining 3,000 data sets.

The commonly used batch sizes are in the 2,000-5,000 range, though smaller or larger batches could provide better performance, depending on the situation. As a result, it is recommended to experiment with different batch sizes to determine the optimum number for a particular object and its use-cases. Note also that you may achieve better performance by also splitting a large bulk operation into multiple transactions (Section 3.5, "Transactions").

For database systems that do not support bulk operations the `bulk` pragma is ignored. It is also possible to specify different batch sizes for different database systems by using the database prefix, for example:

```
#pragma db object mssql:bulk(3000) oracle:bulk(4000) pgsql:bulk(2000)
class person
{
  ...
};
```

Note that while specifying the batch size at compile time might seem inflexible, this approach allows ODB to place internal arrays of the fixed batch size on the stack rather than allocating them in the dynamic memory. However, specifying the batch size at runtime may be supported in the future.

Once the bulk support is enabled for a particular object, we can use the following `database` functions to perform bulk operations:

```
template <typename I>
void
persist (I begin, I end, bool continue_failed = true);

template <typename I>
void
update (I begin, I end, bool continue_failed = true);

template <typename I>
void
erase (I obj_begin, I obj_end, bool continue_failed = true);

template <typename T, typename I>
void
erase (I id_begin, I id_end, bool continue_failed = true);
```

Every bulk API function expects a range of elements, passed in the canonical C++ form as a pair of input iterators. In case of `persist()`, `update()`, and the first `erase()` overload, we pass a range of objects, either as references or as pointers, raw or smart. The following example illustrates the most common scenarios using the `persist()` call:

```
// C array of objects.
//
person a[2] {{"John", "Doe"}, {"Jane", "Doe"}};

db.persist (a, a + sizeof(a) / sizeof(a[0]));


// Vector of objects.
//
std::vector<person> v {{"John", "Doe"}, {"Jane", "Doe"}};

db.persist (v.begin (), v.end ());


// C array of raw pointers to objects.
//
person p1 ("John", "Doe");
person p2 ("Jane", "Doe");
person* pa[2] {&p1, &p2};

db.persist (pa, pa + sizeof(pa) / sizeof(pa[0]));


// Vector of raw pointers to objects.
//
std::vector<person*> pv {&p1, &p2};
```

```
db.persist (pv.begin (), pv.end ());


// Vector of smart (shared) pointers to objects.
//
std::vector<std::shared_ptr<person>> sv {
  std::make_shared<person> ("John", "Doe"),
  std::make_shared<person> ("Jane", "Doe")};

db.persist (sv.begin (), sv.end ());
```

The ability to perform a bulk operation on a range of raw pointers to objects can be especially useful when the application stores objects in a way that does not easily conform to the pair of iterators interface. In such cases we can create a temporary container of shallow pointers to objects and use that to perform the bulk operation, for example:

```
struct person_entry
{
  person obj;

  // Some additional data.
  ...
};

using people = std::vector<person_entry>;

void
persist (odb::database& db, people& p)
{
  std::vector<person*> tmp;
  tmp.reserve (p.size ());
  std::for_each (p.begin (),
                 p.end (),
                 [&tmp] (person_entry& pe)
                 {
                   tmp.push_back (&pe.obj);
                 });


  db.persist (tmp.begin (), tmp.end ());
}
```

The second overload of the bulk `erase()` function allows us to pass a range of object ids rather than objects themselves. As with the corresponding non-bulk version, we have to specify the object type explicitly, for example:

```
std::vector<unsigned long long> ids {1, 2};

db.erase<person> (ids.begin (), ids.end ());
```

Conceptually, a bulk operation is equivalent to performing the corresponding non-bulk version in a loop, except when it comes to the failure semantics. Some databases that currently are capable of bulk operations (specifically, Oracle and SQL Server) do not stop when a data set in a batch fails (for example, because of a unique constraint violation). Instead, they continue executing subsequent data sets until every element in the batch has been attempted. The `continue_failed` argument in the bulk functions listed above specifies whether ODB should extend this behavior and continue with subsequent batches if the one it has tried to execute has failed elements. The default behavior is to continue.

The consequence of this failure semantics is that we may have multiple elements in the range failed for different reasons. For example, if we tried to persist a number of objects, some of them might have failed because they are already persistent while others — because of a unique constraint violation. As a result, ODB uses the special `odb::multiple_exceptions` class to report failures in the bulk API functions. This exception is thrown if one or more elements in the range have failed and it contains the error information in the form of other ODB exception for each failed position. The `multiple_exceptions` class has the following interface:

```
struct multiple_exceptions: odb::exception
{
  // Element type.
  //
  struct value_type
  {
    std::size_t
    position () const;

    const odb::exception&
    exception () const;

    bool
    maybe () const;
  };

  // Iteration.
  //
  using set_type = std::set<value_type>;

  using iterator = set_type::const_iterator;
  using const_iterator = set_type::const_iterator;

  iterator
  begin () const;

  iterator
```

```
  end () const;

  // Lookup.
  //
  const value_type*
  operator[] (std::size_t) const;

  // Severity, failed and attempted counts.
  //
  std::size_t
  attempted () const;

  std::size_t
  failed () const;

  bool
  fatal () const;

  void
  fatal (bool);

  // Direct data access.
  //
  const set_type&
  set () const;

  // odb::exception interface.
  //
  virtual const char*
  what () const throw ();
};
```

The `multiple_exceptions` class has a map-like interface with the key being the position in the range and the value being the exception plus the `maybe` flag (discussed below). As a result, we can either iterate over the failed positions or we can check whether a specific position in the range has failed. The following example shows what a `catch`-handler for this exception might look like:

```
std::vector<person> objs {{"John", "Doe"}, {"Jane", "Doe"}};

try
{
  db.persist (objs.begin (), objs.end ());
}
catch (const odb::multiple_exceptions& me)
{
  for (const auto& v: me)
  {
    size_t p (v.position ());
```

```
    try
    {
      throw v.exception ();
    }
    catch (const odb::object_already_persistent&)
    {
      cerr << p << ": duplicate id: " << objs[p].id () << endl;
    }
    catch (const odb::exception& e)
    {
      cerr << p << ": " << e.what () << endl;
    }
  }
}
```

If, however, all we want is to show the diagnostics to the user, then the string returned by the `what()` function will contain the error information for each failed position. Here is what it might look like (using Oracle as an example):

```
multiple exceptions, 4 elements attempted, 2 failed:
[0] object already persistent
[3] 1: ORA-00001: unique constraint (ODB_TEST.person_last_i) violated
```

Some databases that currently are capable of bulk operations (specifically, Oracle and SQL Server) return a total count of affected rows rather than individual counts for each data set. This limitation prevents ODB from being able to always determine which elements in the batch haven't affected any rows and, for the update and erase operations, translate this to the `object_not_persistent` exceptions. As a result, if some elements in the batch haven't affected any rows and ODB is unable to determine exactly which ones, it will mark all the elements in this batch as "maybe not persistent". That is, it will insert the `object_not_persistent` exception and set the `maybe` flag for every position in the batch. The diagnostics string returned by `what()` will also reflect this situation, for example (assuming batch size of 3):

```
multiple exceptions, 4 elements attempted, 4 failed:
[0-2] (some) object not persistent
[3] object not persistent
```

The way to handle and recover from such "maybe failures" will have to be application-specific. For example, for some applications the fact that some objects no longer exist in the database when performing bulk erase might be an ignorable error. If, however, the application needs to determine exactly which elements in the batch have failed, then a `load()` call will be required for each element in the batch (or a query using a view to avoid loading all the data members; Chapter 10, "Views"). This is also something to keep in mind when selecting the batch size since for larger sizes it will be more expensive (more loads to perform) to handle such "maybe failures". If the failures are not uncommon, as is the case, for example, when using optimistic concurrency, then it may make sense to use a smaller batch.

The lookup operator (`operator[]`) returns `NULL` if the element at this position has no exception. Note also that the returned value is `value_type*` and not `odb::exception*` in order to provide access to the `maybe` flag discussed above.

The `multiple_exceptions` class also provides access to the number of positions attempted (the `attempted()` accessor) and failed (the `failed()` accessor). Note that the failed count includes the "maybe failed" positions.

The `multiple_exceptions` exception can also be fatal. If the `fatal()` accessor returns `true`, then (some of) the exceptions were fatal. In this case, even for positions that did not fail, no attempts were made to complete the operation and the transaction must be aborted.

If `fatal()` returns false, then the operation on the elements that don't have an exception has succeeded. The application can ignore the errors or try to correct the errors and re-attempt the operation on the elements that did fail. In either case, the transaction can be committed.

An example of a fatal exception would be the situation where the execution of the underlying statement failed summarily, without attempting any data sets, for instance, because of an error in the statement itself.

The `fatal()` modifier allows you to "upgrade" an exception to fatal, for example, for specific database error codes.

PART II DATABASE SYSTEMS

Part II covers topics specific to the database system implementations and their support in ODB. The first chapter in Part II discusses how to use multiple database systems in the same application. The subsequent chapters describe the system-specific `database` classes as well as the default mapping between basic C++ value types and native database types. Part II consists of the following chapters.

16 Multi-Database Support

17 MySQL Database

18 SQLite Database

19 PostgreSQL Database

20 Oracle Database

21 Microsoft SQL Server Database

16 Multi-Database Support

Some applications may need to access multiple database systems, either simultaneously or one at a time. For example, an application may utilize an embedded database such as SQLite as a local cache and use a client-server database such as PostgreSQL for more permanent but slower to access remote storage. Or an application may need to be able to store its data in any database selected at runtime by the user. Yet another scenario is the data migration from one database system to another. In this case, multi-database support is only required for a short period. It is also plausible that an application implements all three of these scenarios, that is, it uses SQLite as a local cache, allows the user to select the remote database system, and supports data migration from one remote database system to another.

ODB provides two types of multi-database support: *static* and *dynamic*. With static support we use the database system-specific interfaces to perform database operations. That is, instead of using `odb::database`, `odb::transaction`, or `odb::query`, we would use, for example, `odb::sqlite::database`, `odb::sqlite::transaction`, or `odb::sqlite::query` to access an SQLite database.

In contrast, with *dynamic* multi-database support we can use the common interface to access any database without having to know which one it is. At runtime, ODB will automatically dispatch a call on the common interface to the specific database implementation based on the actual `database` instance being used. In fact, this mechanism is very similar to C++ virtual functions.

Both static and dynamic multi-database support have a different set of advantages and disadvantages which makes them more or less suitable for different use cases. Static support has zero overhead compared to single-database support and allows us to use database system-specific features, extensions, etc. At the same time, the code that we write will be tied to the specific database system. As a result, this type of multi-database support is more suitable for situations where different parts of an application access different but specific database systems. For example, using SQLite as a local cache most likely falls into this category since we are using a specific database system (SQLite) and the code that will check the cache will most likely (but not necessarily) be separate from the code that interact with the remote database. Another example where static multi-database support might be more suitable is a once-off data migration from one database system to another. In this case both the source and target are specific database systems. In contrast, if data migration from one database system to another is a general feature in an application, then dynamic multi-database support might be more suitable.

The main advantage of dynamic multi-database support is the database system-independence of the code that we write. The same application code will work with any database system supported by ODB and the generated database support code can be packaged into separate libraries and loaded dynamically by the application. The disadvantages of dynamic support are slight overhead and certain limitations in functionality compared to static support (see Section 16.2, "Dynamic Multi-Database Support" for details). As a result, dynamic multi-database support is most suitable

to situations where we need the same code to work with a range of database systems. For example, if your application must be able to store its data in any database selected by the user, then dynamic support is probably the best option.

Note also that it is possible to mix and match static and dynamic support in the same application. In fact, dynamic support is built on top of static support so it is possible to use the same database system both "statically" and "dynamically". In particular, the ability to "drop down" from dynamic to static support can be used to overcome the functionality limitations mentioned above. Finally, single-database support is just a special case of static multi-database support with a single database system.

By default ODB assumes single-database support. To enable multi-database support we use the `--multi-database` (or `-m`) ODB compiler option. This option is also used to specify the support type: `static` or `dynamic`. For example:

```
odb -m static ... person.hxx
```

With multi-database support enabled, we can now generate the database support code for several database systems. This can be accomplished either with a single ODB compiler invocation by specifying multiple `--database` (or `-d`) options or with multiple ODB compiler invocations. Both approaches produce the same result, for example:

```
odb -m static -d common -d sqlite -d pgsql person.hxx
```

Is equivalent to:

```
odb -m static -d common person.hxx
odb -m static -d sqlite person.hxx
odb -m static -d pgsql person.hxx
```

Notice that the first `-d` option has `common` as its value. This is not a real database system. Rather, it instructs the ODB compiler to generate code that is common to all the database systems and, in case of dynamic support, is also the common interfaces.

If you look at the result of the above commands, you will also notice changes in the output file names. In the single-database mode the ODB compiler produces a single set of the `person-odb.?xx` files which contain both the common as well as the database specific generated code (since there is only one database system in use, there is no reason to split the two). In contrast, in the multi-database mode, the `person-odb.?xx` set of files contains the common code while the database system-specific code is written to files in the form `person-odb-<db>.?xx`. That is, `person-odb-sqlite.?xx` for SQLite, `person-odb-pgsql.?xx` for PostgreSQL, etc.

If we need dynamic support for some databases and static for others, then the `common` code must be generated in the dynamic mode. For example, if we need static support for SQLite and dynamic support for PostgreSQL and Oracle, then the ODB compiler invocations could look like this:

```
odb -m dynamic -d common person.hxx
odb -m static -d sqlite person.hxx
odb -m dynamic -d pgsql person.hxx
odb -m dynamic -d oracle person.hxx
```

With multi-database support enabled, it is possible to restrict ODB pragmas to apply only to a specific database system (unrestricted pragmas apply to all the databases). For example:

```
#pragma db object
class person
{
  ...

  #pragma db pgsql:type("VARCHAR(128)") sqlite:type("TEXT")
  std::string name_;

  unsigned short age_;

  #pragma db pgsql index member(age_)
};
```

Above, the pragma for the `name_` data member shows the use of a database prefix (for example, `pgsql:`) that only applies to the specifier that follows. The pragma that defines an index on the `age_` data member shows the use of a database prefix that applies to the whole pragma. In this case the database name must immediately follow the `db` keyword.

Similar to pragmas, ODB compiler options that determine the kind (for example, `--schema-format`), names (for example, `--odb-file-suffix`), or content (for example, prologue and epilogue options) of the output files can be prefixed with the database name. For example:

```
odb --odb-file-suffix common:-odb-common ...
```

Dynamic multi-database support requires consistent mapping across all the databases. That is, the same classes and data members should be mapped to objects, simple/composite values, etc., for all the databases. In contrast, static multi-database support does not have this restriction. Specifically, with static support, some data members can be transient for some database systems. Similarly, the same class (for example, `point`) can be mapped to a simple value in one database (for example, to the `POINT` PostgreSQL type) and to a composite value in another (for example, in SQLite, which does not have a built-in point type).

The following sections discuss static and dynamic multi-database support in more detail.

16.1 Static Multi-Database Support

With static multi-database support, instead of including `person-odb.hxx`, application source code has to include `person-odb-<db>.hxx` header files corresponding to the database systems that will be used.

The application code has to also use database system-specific interfaces when performing database operations. As an example, consider the following transaction in a single-database application. It uses the common interfaces, that is, classes from the `odb` namespace.

```
#include "person-odb.hxx"

odb::database& db = ...

using query = odb::query<person>;
using result = odb::result<person>;

odb::transaction t (db.begin ());
result r (db.query<person> (query::age < 30));
...
t.commit ();
```

In an application that employs static multi-database support the same transaction for SQLite would be rewritten like this:

```
#include "person-odb-sqlite.hxx"

odb::sqlite::database& db = ...

using query = odb::sqlite::query<person>;
using result = odb::result<person>;      // odb:: not odb::sqlite::

odb::sqlite::transaction t (db.begin ());
result r (db.query<person> (query::age < 30));
...
t.commit ();
```

That is, the `database`, `transaction`, and `query` classes now come from the `odb::sqlite` namespace instead of `odb`. Other classes that have database system-specific interfaces are `connection`, `statement`, and `tracer`. Note that all of them derive from the corresponding common versions. It is also possible to use common `transaction`, `connection`, and `statement` classes with static support, if desired.

Notice that we didn't use the `odb::sqlite` namespace for the `result` class template. This is because `result` is database system-independent. All other classes defined in namespace `odb`, except those specifically mentioned above, are database system-independent. In particular, `result`, `prepared_query`, `session`, `schema_catalog`, and all the exceptions are database system-independent.

Writing `odb::sqlite::` before every name can quickly become burdensome. As we have seen before, in single-database applications that use the common interface we can add the `using namespace` directive to avoid qualifying each name. For example:

```
#include "person-odb.hxx"

odb::database& db = ...

{
  using namespace odb::core;

  using person_query = query<person>;
  using person_result = result<person>;

  transaction t (db.begin ());
  person_result r (db.query<person> (person_query::age < 30));
  ...
  t.commit ();
}
```

A similar mechanism is available in multi-database support. Each database runtime defines the `odb::<db>::core` namespace that contains all the database system-independent names as well as the database system-specific ones for this database. Here is how we can rewire the above transaction using this approach:

```
#include "person-odb-sqlite.hxx"

odb::sqlite::database& db = ...

{
  using namespace odb::sqlite::core;

  using person_query = query<person>;
  using person_result = result<person>;

  transaction t (db.begin ());
  person_result r (db.query<person> (person_query::age < 30));
  ...
  t.commit ();
}
```

If the `using namespace` directive cannot be used, for example, because the same code fragment accesses several databases, then we can still make the namespace qualifications more concise by assigning shorter aliases to database namespaces. For example:

```
#include "person-odb-pgsql.hxx"
#include "person-odb-sqlite.hxx"

namespace pg = odb::pgsql;
namespace sl = odb::sqlite;

pg::database& pg_db = ...
sl::database& sl_db = ...

using pg_query = pg::query<person>;
using sl_query = sl::query<person>;
using result = odb::result<person>;

// First check the local cache.
//
odb::transaction t (sl_db.begin ()); // Note: using common transaction.
result r (sl_db.query<person> (sl_query::age < 30));

// If no hits, try the remote database.
//
if (r.empty ())
{
  t.commit ();               // End the SQLite transaction.
  t.reset (pg_db.begin ()); // Start the PostgreSQL transaction.

  r = pg_db.query<person> (pg_query::age < 30);
}

// Handle the result.
//
...

t.commit ();
```

With static multi-database support we can make one of the databases the default database with the `--default-database` option. The default database can be accessed via the common interface, just like with single-database support. For example:

```
odb -m static -d common -d pgsql -d sqlite --default-database pgsql ...
```

The default database mechanism can be useful when one of the databases is primary or when retrofitting multi-database support into an existing single-database application. For example, if we are adding SQLite as a local cache into an existing application that uses PostgreSQL as its only database, then by making PostgreSQL the default database we avoid having to change all the existing code. Note that if dynamic multi-database support is enabled, then the common

(dynamic) interface is always made the default database.

16.2 Dynamic Multi-Database Support

With dynamic multi-database support, application source code only needs to include the `person-odb.hxx` header file, just like with single-database support. In particular, we don't need to include any of the `person-odb-<db>.hxx` files unless we would also like to use certain database systems in the static multi-database mode.

When performing database operations, the application code uses the common interfaces from the `odb` namespace, just like with single-database support. As an example, consider a function that can be used to load an object either from a local SQLite cache or a remote PostgreSQL database (in reality, this function can be used with any database system support by ODB provided we generated the database support code for this database and linked it into our application):

```
#include "person-odb.hxx"

std::unique_ptr<person>
load (odb::database& db, const std::string& name)
{
  odb::transaction t (db.begin ());
  std::unique_ptr<person> p (db.find (name));
  t.commit ();
  return p;
}

odb::pgsql::database& pg_db = ...
odb::sqlite::database& sl_db = ...

// First try the local cache.
//
std::unique_ptr<person> p (load (sl_db, "John Doe"));

// If not found, try the remote database.
//
if (p == 0)
  p = load (pg_db, "John Doe");

...
```

As you can see, we can use dynamic multi-database support just like single-database support except that now our code can work with different database systems. Note, however, one difference: with single-database support we could perform database operations using either the common `odb::database` or a database system-specific (for example, `odb::sqlite::database`) interface with the same effect. In contrast, with dynamic multi-database support, the use of the database system-specific interface results in the switch to the static mode (for which, as was mentioned earlier, we would need to include the corresponding

`person-odb-<db>.hxx` header file). As we will discuss shortly, switching from dynamic to static mode can be used to overcome limitations imposed by dynamic multi-database support.

Dynamic multi-database support has certain overheads and limitations compared to static support. For database operations, the generated code maintains function tables that are used to dispatch calls to the database system-specific implementations. In single-database and static multi-database support, the `query` type implements a thin wrapper around the underlying database system's `SELECT` statement. With dynamic multi-database support, because the underlying database system is only known at query execution (or preparation) time, the `query` type stores a database system-independent representation of the query that is then translated to the database system-specific form. Because of this database system-independent representation, dynamic support queries have a number of limitations. Specifically, dynamic queries do not support parameter binding in native query fragments. They also make copies of by-value parameterd (by-reference parameters can be used to remove this overhead). Finally, parameters of array types (for example, `char[256]`) can only be bound by-reference.

As we mentioned earlier, switching from dynamic to static mode can be an effective way to overcome these limitations. As an example, consider a function that prints the list of people of a certain age. The caller also specified the limit on the number of entries to print. Some database systems, for example, PostgreSQL, allow us to propagate this limit to the database server with the `LIMIT` clause. To add this clause we would need to construct a native query fragment and, as we discussed above, we won't be able to bind a parameter (the limit) while in the dynamic mode. The following implementation shows how we can overcome this by switching to the static mode and using the PostgreSQL-specific interface:

```
#include "person-odb.hxx"
#include "person-odb-pgsql.hxx" // Needed for static mode.

void
print (odb::database& db, unsigned short age, unsigned long limit)
{
  using query = odb::query<person>;
  using result = odb::result<person>;

  odb::transaction t (db.begin ());

  query q (query::age == age);
  result r;

  if (db.id () == odb::id_pgsql)
  {
    // We are using PostgreSQL. Drop down to the static mode and
    // add the LIMIT clause to the query.
    //
    namespace pg = odb::pgsql;
    using pg_query = pg::query<person>;
```

```
    pg::database& pg_db (static_cast<pg::database&> (db));
    pg_query pg_q (pg_query (q) + "LIMIT" + pg_query::_val (limit));
    r = pg_db.query<person> (pg_q);
  }
  else
    r = db.query<person> (q);

  // Handle the result up to the limit elements.
  //
  ...

  t.commit ();
}

odb::pgsql::database& pg_db = ...
odb::sqlite::database& sl_db = ...

print (sl_db, 30, 100);
print (sl_db, 30, 100);
```

A few things to note about this example. First, we use the `database::id()` function to determine the actual database system we use. This function has the following signature:

```
namespace odb
{
  enum database_id
  {
    id_mysql,
    id_sqlite,
    id_pgsql,
    id_oracle,
    id_mssql,
    id_common
  };

  class database
  {
  public:
    ...

    database_id
    id () const;
  }
}
```

Note that `database::id()` can never return the `id_common` value.

The other thing to note is how we translate the dynamic query to the database system-specific one (the `pg_query (q)` expression). Every `odb::<db>::query` class provides such a translation constructor.

16.2.2 Dynamic Loading of Database Support Code

With dynamic multi-database support, the generated database support code automatically registers itself with the function tables that we mentioned earlier. This makes it possible to package the generated code for each database into a separate dynamic-link library (Windows DLL) or dynamic shared object (Unix DSO; collectively referred to as DLLs from now on) and load/unload them from the application dynamically using APIs such as Win32 `LoadLibrary()` or POSIX `dlopen()`. This allows the application address space to contain code only for database systems that are actually needed in any particular moment. Another advantage of this approach is the ability to distribute individual database system support separately.

This section provides an overview of how to package the generated database support code into DLLs for both Windows and Unix using GNU/Linux as an example. Note also that if static multi-database support is used for a particular database system, then the dynamic loading cannot be used for this database. It is, however, still possible to package the generated code into a DLL but this DLL will have to be linked to the executable at link-time rather than at runtime. If dynamic loading is desirable in this situation, then another alternative would be to package the functionality that requires static support together with the database support code into the DLL and import this functionality dynamically using the `GetProcAddress()` (Win32) or `dlsym()` (Unix) function.

The first step in packaging the generated code into DLLs is to set up the symbol exporting. This step is required for Windows DLLs but is optional for Unix DSOs. Most modern Unix systems (such as GNU/Linux) provide control over symbol visibility, which is a mechanism similar to Windows symbol exporting. Notable advantages of using this mechanism to explicitly specify which symbols are visible include smaller Unix DSOs and faster load times. If, however, you are not planning to control symbol visibility on Unix, then you can skip directly to the second step below.

An important point to understand is that we only need to export the common interface, that is, the classes defined in the `person-odb.hxx` header. In particular, we don't need to export the database system-specific classes defined in the `person-odb-<db>.hxx`, unless we are also using this database in the static mode (in which case, the procedure described below will need to be repeated for that database as well).

The ODB compiler provides two command line options, `--export-symbol` and `--extern-symbol`, which can be used to insert the export and extern macros in all the necessary places in the generated header file. You are probably familiar with the concept of export macro which expands to an export directive if we are building the DLL and to an import directive if we are building client code. The extern macro is a supplementary mechanism which is necessary to export explicit template instantiations used by the generated code when query support is enabled. As we will see shortly, the extern macro must expand into the `extern` C++ keyword in certain situations and must be left undefined in others. To manage all these macro definitions, it is

customary to create the so called export header. Based on a single macro that is normally defined in the project file or on the command line and which indicates whether we are building the DLL or client code, the export header file sets the export and extern macros to their appropriate values. Continuing with our person example, on Windows the export header, which we will call `person-export.hxx`, could look like this:

```
// person-export.hxx
//
// Define PERSON_BUILD_DLL if we are building the DLL. Leave it
// undefined in client code.
//
#ifndef PERSON_EXPORT_HXX
#define PERSON_EXPORT_HXX

#ifdef PERSON_BUILD_DLL
#  define PERSON_EXPORT __declspec(dllexport)
#else
#  define PERSON_EXPORT __declspec(dllimport)
#  define PERSON_EXTERN extern
#endif

#endif // PERSON_EXPORT_HXX
```

The equivalent export header for GCC on GNU/Linux is shown below. Note also that on GNU/Linux, by default, all symbols are visible and we need to add the GCC `-fvisibility=hidden` option to make them hidden by default.

```
// person-export.hxx
//
#ifndef PERSON_EXPORT_HXX
#define PERSON_EXPORT_HXX

#define PERSON_EXPORT __attribute__ ((visibility ("default")))
#define PERSON_EXTERN extern

#endif // PERSON_EXPORT_HXX
```

Next we need to export the `person` persistent class using the export macro and re-compile our `person.hxx` file with the `--export-symbol` and `--extern-symbol` options. We will also need to include `person-export.hxx` into the generated `person-odb.hxx` file. For that we use the `--hxx-prologue` option. Here is how we can do this with multiple invocations of the ODB compiler:

```
odb -m dynamic -d common --hxx-prologue "#include \"person-export.hxx\"" \
--export-symbol PERSON_EXPORT --extern-symbol PERSON_EXTERN person.hxx

odb -m dynamic -d sqlite person.hxx
odb -m dynamic -d pgsql person.hxx
```

It is also possible to achieve the same with a single invocation. Here we need to restrict some option values to apply only to the `common` database:

```
odb -m dynamic -d common -d sqlite -d pgsql \
--hxx-prologue "common:#include \"person-export.hxx\"" \
--export-symbol common:PERSON_EXPORT --extern-symbol common:PERSON_EXTERN \
person.hxx
```

The second step in packaging the generated code into DLLs is to decide where to place the generated common interface code. One option is to place it into a DLL of its own so that we will end up with (replace `*.dll` with `lib*.so` for Unix): `person.dll` plus `person-sqlite.dll` and `person-pgsql.dll`, which both link to `person.dll`, as well as `person.exe`, which links to `person.dll` and dynamically loads `person-sqlite.dll` and/or `person-pgsql.dll`. If this is the organization that you prefer, then the next step is to build all the DLLs as you normally would any other DLL, placing `person-odb.cxx` and `person.cxx` into `person.dll`, `person-odb-sqlite.cxx` into `person-sqlite.dll`, etc. Note that in the pure dynamic multi-database support, `person-sqlite.dll` and `person-pgsql.dll` do not export any symbols.

We can improve on the above organization by getting rid of `person.dll`, which is not really necessary unless we have multiple executables sharing the same database support. To achieve this, we will place `person-odb.cxx` into `person.exe` and export its symbols from the executable instead of a DLL. Exporting symbols from an executable is a seldom used functionality, especially on Windows, however, it is well supported on both Windows and most Unix platforms. Note also that this approach won't work if we also use one of the databases in the static mode.

On Windows all we have to do is place `person-odb.cxx` into the executable and compile it as we would in a DLL (that is, with the `PERSON_BUILD_DLL` macro defined). If Windows linker detects that an executable exports any symbols, then it will automatically create the corresponding import library (`person.lib` in our case). We then use this import library to build `person-sqlite.dll` and `person-pgsql.dll` as before.

To export symbols from an executable on GNU/Linux all we need to do is add the `-rdynamic` option when linking our executable.

17 MySQL Database

To generate support code for the MySQL database you will need to pass the "`--database mysql`" (or "`-d mysql`") option to the ODB compiler. Your application will also need to link to the MySQL ODB runtime library (`libodb-mysql`). All MySQL-specific ODB classes are defined in the `odb::mysql` namespace.

17.1 MySQL Type Mapping

The following table summarizes the default mapping between basic C++ value types and MySQL database types. This mapping can be customized on the per-type and per-member basis using the ODB Pragma Language (Chapter 14, "ODB Pragma Language").

C++ Type	MySQL Type	Default `NULL` Semantics
`bool`	`TINYINT(1)`	`NOT NULL`
`char`	`CHAR(1)`	`NOT NULL`
`signed char`	`TINYINT`	`NOT NULL`
`unsigned char`	`TINYINT UNSIGNED`	`NOT NULL`
`short`	`SMALLINT`	`NOT NULL`
`unsigned short`	`SMALLINT UNSIGNED`	`NOT NULL`
`int`	`INT`	`NOT NULL`
`unsigned int`	`INT UNSIGNED`	`NOT NULL`
`long`	`BIGINT`	`NOT NULL`
`unsigned long`	`BIGINT UNSIGNED`	`NOT NULL`
`long long`	`BIGINT`	`NOT NULL`
`unsigned long long`	`BIGINT UNSIGNED`	`NOT NULL`
`float`	`FLOAT`	`NOT NULL`
`double`	`DOUBLE`	`NOT NULL`
`std::string`	`TEXT/VARCHAR(128)`	`NOT NULL`
`char[N]`	`VARCHAR(N-1)`	`NOT NULL`

It is possible to map the `char` C++ type to an integer database type (for example, `TINYINT`) using the `db type` pragma (Section 14.4.3, "`type`").

Note that the `std::string` type is mapped differently depending on whether a member of this type is an object id or not. If the member is an object id, then for this member `std::string` is mapped to the `VARCHAR(128)` MySQL type. Otherwise, it is mapped to `TEXT`.

Additionally, by default, C++ enums and C++11 enum classes are automatically mapped to suitable MySQL types. Contiguous enumerations with the zero first enumerator are mapped to the MySQL `ENUM` type. All other enumerations are mapped to the MySQL types corresponding to their underlying integral types (see table above). In both cases the default `NULL` semantics is `NOT NULL`. For example:

```
enum color {red, green, blue};
enum class taste: unsigned char
{
  bitter = 1, // Non-zero first enumerator.
  sweet,
  sour = 4,   // Non-contiguous.
  salty
};

#pragma db object
class object
{
  ...

  color color_; // Mapped to ENUM ('red', 'green', 'blue') NOT NULL.
  taste taste_; // Mapped to TINYNT UNSIGNED NOT NULL.
};
```

The only built-in mapping provided for the MySQL `DECIMAL` type is to `std::string/char[N]`, for example:

```
#pragma db object
class object
{
  ...

  #pragma db type ("DECIMAL(6,3)")
  std::string value_;
};
```

You can, however, map `DECIMAL` to a custom C++ type by providing a suitable `odb::mysql::value_traits` specialization.

It is also possible to add support for additional MySQL types, such as geospatial types. For more information, refer to Section 14.8, "Database Type Mapping Pragmas".

17.1.1 String Type Mapping

The MySQL ODB runtime library provides support for mapping the `std::string`, `char[N]`, and `std::array<char, N>` types to the MySQL `CHAR`, `VARCHAR`, `TEXT`, `NCHAR`, and `NVARCHAR` types. However, these mappings are not enabled by default (in particular, by default, `std::array` will be treated as a container). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("CHAR(2)")
  char state_[2];

  #pragma db type("VARCHAR(128)")
  std::string name_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
#pragma db value(std::string) type("VARCHAR(128)")

#pragma db object
class object
{
  ...

  std::string name_; // Mapped to VARCHAR(128).
};
```

The `char[N]` and `std::array<char, N>` values may or may not be zero-terminated. When extracting such values from the database, ODB will append the zero terminator if there is enough space.

17.1.2 Binary Type Mapping

The MySQL ODB runtime library provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, `unsigned char[N]`, `std::array<char, N>`, and `std::array<unsigned char, N>` types to the MySQL `BINARY`, `VARBINARY`, and `BLOB` types. However, these mappings are not enabled by default (in particular, by default, `std::vector` and `std::array` will be treated as containers). To

enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("BINARY(16)")
  unsigned char uuid_[16];
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using buffer = std::vector<char>;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Note also that in native queries (Chapter 4, "Querying the Database") `char[N]` and `std::array<char, N>` parameters are by default passed as a string rather than a binary. To pass such parameters as a binary, we need to specify the database type explicitly in the `_val()`/`_ref()` calls. Note also that we don't need to do this for the integrated queries, for example:

```
char u[16] = {...};

db.query<object> ("uuid = " + query::_val<odb::mysql::id_blob> (u));
db.query<object> (query::uuid == query::_ref (u));
```

17.1.3 Mixed Automatic/0 Object Id Assignment

In MySQL an automatic object id can also be set manually to `0`. For example:

```
  #pragma db id auto
  odb::nullable<int64_t> id;
```

Then, when used with the `NO_AUTO_VALUE_ON_ZERO` mode, set the id member to `NULL` to get auto-assignment or to `0` to use `0` as the id. This functionality is normally used to assign the special `0` id to a special object.

17.2 MySQL Database Class

The MySQL `database` class has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class database: public odb::database
    {
    public:
      database (const char* user,
                const char* passwd,
                const char* db,
                const char* host = 0,
                unsigned int port = 0,
                const char* socket = 0,
                const char* charset = 0,
                unsigned long client_flags = 0,
                std::[auto|unique]_ptr<connection_factory> = 0);

      database (const std::string& user,
                const std::string& passwd,
                const std::string& db,
                const std::string& host = "",
                unsigned int port = 0,
                const std::string* socket = 0,
                const std::string& charset = "",
                unsigned long client_flags = 0,
                std::[auto|unique]_ptr<connection_factory> = 0);

      database (const std::string& user,
                const std::string* passwd,
                const std::string& db,
                const std::string& host = "",
                unsigned int port = 0,
                const std::string* socket = 0,
                const std::string& charset = "",
                unsigned long client_flags = 0,
                std::[auto|unique]_ptr<connection_factory> = 0);

      database (const std::string& user,
                const std::string& passwd,
                const std::string& db,
                const std::string& host,
                unsigned int port,
```

```
            const std::string& socket,
            const std::string& charset = "",
            unsigned long client_flags = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  database (const std::string& user,
            const std::string* passwd,
            const std::string& db,
            const std::string& host,
            unsigned int port,
            const std::string& socket,
            const std::string& charset = "",
            unsigned long client_flags = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  database (int& argc,
            char* argv[],
            bool erase = false,
            const std::string& charset = "",
            unsigned long client_flags = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  static void
  print_usage (std::ostream&);

public:
  const char*
  user () const;

  const char*
  password () const;

  const char*
  db () const;

  const char*
  host () const;

  unsigned int
  port () const;

  const char*
  socket () const;

  const char*
  charset () const;

  unsigned long
  client_flags () const;

public:
```

```
      connection_ptr
      connection ();
    };
  }
}
```

You will need to include the `<odb/mysql/database.hxx>` header file to make this class available in your application.

The overloaded `database` constructors allow us to specify MySQL database parameters that should be used when connecting to the database. In MySQL `NULL` and an empty string are treated as the same values for all the string parameters except `password` and `socket`.

The `charset` argument allows us to specify the client character set, that is, the character set in which the application will encode its text data. Note that this can be different from the MySQL server character set. If this argument is not specified or is empty, then the default MySQL client character set is used, normally `latin1`. Commonly used values for this argument are `latin1` (equivalent to Windows cp1252 and similar to ISO-8859-1) and `utf8`. For other possible values as well as more information on character set support in MySQL, refer to the MySQL documentation.

The `client_flags` argument allows us to specify various MySQL client library flags. For more information on the possible values, refer to the MySQL C API documentation. The `CLIENT_FOUND_ROWS` flag is always set by the MySQL ODB runtime regardless of whether it was passed in the `client_flags` argument.

The last constructor extracts the database parameters from the command line. The following options are recognized:

```
--user <login>
--password <password>
--database <name>
--host <host>
--port <integer>
--socket <socket>
--options-file <file>
```

The `--options-file` option allows us to specify some or all of the database options in a file with each option appearing on a separate line followed by a space and an option value.

If the `erase` argument to this constructor is true, then the above options are removed from the `argv` array and the `argc` count is updated accordingly. This is primarily useful if your application accepts other options or arguments and you would like to get the MySQL options out of the `argv` array.

This constructor throws the `odb::mysql::cli_exception` exception if the MySQL option values are missing or invalid. See section Section 17.4, "MySQL Exceptions" for more information on this exception.

The static `print_usage()` function prints the list of options with short descriptions that are recognized by this constructor.

The last argument to all of the constructors is a pointer to the connection factory. In C++98/03, it is `std::auto_ptr` while in C++11 `std::unique_ptr` is used instead. If we pass a non-`NULL` value, the database instance assumes ownership of the factory instance. The connection factory interface as well as the available implementations are described in the next section.

The set of accessor functions following the constructors allows us to query the parameters of the `database` instance.

The `connection()` function returns a pointer to the MySQL database connection encapsulated by the `odb::mysql::connection` class. For more information on `mysql::connection`, refer to Section 17.3, "MySQL Connection and Connection Factory".

17.3 MySQL Connection and Connection Factory

The `mysql::connection` class has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class connection: public odb::connection
    {
    public:
      connection (database&);
      connection (database&, MYSQL*);

      MYSQL*
      handle ();
    };

    using connection_ptr = details::shared_ptr<connection>;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.6, "Connections". The first overloaded `mysql::connection` constructor establishes a new MySQL connection. The second constructor allows us to create a `connection` instance by providing an already connected native MySQL handle. Note that the `connection` instance assumes ownership of this handle. The `handle()` accessor returns the MySQL handle corresponding to the connec-

tion.

The `mysql::connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class connection_factory
    {
    public:
      virtual void
      database (database&) = 0;

      virtual connection_ptr
      connect () = 0;
    };
  }
}
```

The `database()` function is called when a connection factory is associated with a database instance. This happens in the `odb::mysql::database` class constructors. The `connect()` function is called whenever a database connection is requested.

The two implementations of the `connection_factory` interface provided by the MySQL ODB runtime are `new_connection_factory` and `connection_pool_factory`. You will need to include the `<odb/mysql/connection-factory.hxx>` header file to make the `connection_factory` interface and these implementation classes available in your application.

The `new_connection_factory` class creates a new connection whenever one is requested. When a connection is no longer needed, it is released and closed. The `new_connection_factory` class has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class new_connection_factory: public connection_factory
    {
    public:
      new_connection_factory ();
    };
};
```

The `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class connection_pool_factory: public connection_factory
    {
    public:
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0,
                               bool ping = true);

    protected:
      class pooled_connection: public connection
      {
      public:
        pooled_connection (database_type&);
        pooled_connection (database_type&, MYSQL*);
      };

      using pooled_connection_ptr = details::shared_ptr<pooled_connection>;

      virtual pooled_connection_ptr
      create ();
    };
};
```

The `max_connections` argument in the `connection_pool_factory` constructor specifies the maximum number of concurrent connections that this pool factory will maintain. Similarly, the `min_connections` argument specifies the minimum number of available connections that should be kept open. The `ping` argument specifies whether the factory should validate the connection before returning it to the caller.

Whenever a connection is requested, the pool factory first checks if there is an unused connection that can be returned. If there is none, the pool factory checks the `max_connections` value to see if a new connection can be created. If the total number of connections maintained by the pool is less than this value, then a new connection is created and returned. Otherwise, the caller is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked callers waiting for a connection. If so, then one of them is unblocked and is given the connection. Otherwise, the pool factory checks whether the total number of connections maintained by the pool is greater than the `min_connections` value. If that's the case, the connection is closed. Otherwise, the connection is added to the pool of available connections to be returned on the next request. In other words, if the number of connections maintained by the pool exceeds `min_connections` and there are no callers waiting for a new connection, then the pool will close the excess connections.

If the `max_connections` value is 0, then the pool will create a new connection whenever all of the existing connections are in use. If the `min_connections` value is 0, then the pool will never close a connection and instead maintain all the connections that were ever created.

Connection validation (the `ping` argument) is useful if your application may experience long periods of inactivity. In such cases the MySQL server may close network connections that have been inactive for too long. If during connection validation the pool factory detects that the connection has been terminated, it silently closes it and tries to find or create another connection instead.

The `create()` virtual function is called whenever the pool needs to create a new connection. By deriving from the `connection_pool_factory` class and overriding this function we can implement custom connection establishment and configuration.

If you pass `NULL` as the connection factory to one of the `database` constructors, then the `connection_pool_factory` instance will be created by default with the min and max connections values set to `0` and connection validation enabled. The following code fragment shows how we can pass our own connection factory instance:

```
#include <odb/database.hxx>

#include <odb/mysql/database.hxx>
#include <odb/mysql/connection-factory.hxx>

int
main (int argc, char* argv[])
{
  unique_ptr<odb::mysql::connection_factory> f (
    new odb::mysql::connection_pool_factory (20));

  unique_ptr<odb::database> db (
    new mysql::database (argc, argv, false, 0, f));
}
```

17.4 MySQL Exceptions

The MySQL ODB runtime library defines the following MySQL-specific exceptions:

```
namespace odb
{
  namespace mysql
  {
    class database_exception: odb::database_exception
    {
    public:
      unsigned int
      error () const;
```

```
      const std::string&
      sqlstate () const;

      const std::string&
      message () const;

      virtual const char*
      what () const throw ();
    };

    class cli_exception: odb::exception
    {
    public:
      virtual const char*
      what () const throw ();
    };
  }
}
```

You will need to include the `<odb/mysql/exceptions.hxx>` header file to make these exceptions available in your application.

The `odb::mysql::database_exception` is thrown if a MySQL database operation fails. The MySQL-specific error information is accessible via the `error()`, `sqlstate()`, and `message()` functions. All this information is also combined and returned in a human-readable form by the `what()` function.

The `odb::mysql::cli_exception` is thrown by the command line parsing constructor of the `odb::mysql::database` class if the MySQL option values are missing or invalid. The `what()` function returns a human-readable description of an error.

17.5 MySQL Limitations

The following sections describe MySQL-specific limitations imposed by the current MySQL and ODB runtime versions.

17.5.1 Foreign Key Constraints

ODB relies on standard SQL behavior which requires that foreign key constraints checking is deferred until the transaction is committed. The only behaviors supported by MySQL are to either check such constraints immediately (InnoDB engine) or to ignore foreign key constraints altogether (all other engines). As a result, by default, schemas generated by the ODB compiler for MySQL have foreign key definitions commented out. They are retained only for documentation.

You can override the default behavior and instruct the ODB compiler to generate non-deferrable foreign keys by specifying the `--fkeys-deferrable-mode not_deferrable` ODB compiler option. Note, however, that in this case the order in which you persist, update, and erase objects within a transaction becomes important.

17.6 MySQL Index Definitions

When the `index` pragma (Section 14.7, "Index Definition Pragmas") is used to define a MySQL index, the `type` clause specifies the index type (for example, `UNIQUE`, `FULLTEXT`, `SPATIAL`), the `method` clause specifies the index method (for example, `BTREE`, `HASH`), and the `options` clause is not used. The column options can be used to specify column length limits and the sort order. For example:

```
#pragma db object
class object
{
  ...

  std::string name_;

  #pragma db index method("HASH") member(name_, "(100) DESC")
};
```

17.7 MySQL Stored Procedures

ODB native views (Section 10.6, "Native Views") can be used to call MySQL stored procedures. For example, assuming we are using the `person` class from Chapter 2, "Hello World Example" (and the corresponding `person` table), we can create a stored procedure that given the min and max ages returns some information about all the people in that range:

```
CREATE PROCEDURE person_range (
  IN min_age SMALLINT,
  IN max_age SMALLINT)
BEGIN
  SELECT age, first, last FROM person
    WHERE age >= min_age AND age <= max_age;
END
```

Given the above stored procedure we can then define an ODB view that can be used to call it and retrieve its result:

```
#pragma db view query("CALL person_range((?))")
struct person_range
{
  unsigned short age;
  std::string first;
  std::string last;
};
```

The following example shows how we can use the above view to print the list of people in a specific age range:

```
using query = odb::query<person_range>;
using result = odb::result<person_range>;

transaction t (db.begin ());

result r (
  db.query<person_range> (
    query::_val (1) + "," + query::_val (18)));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cerr << i->first << " " << i->last << " " << i->age << endl;

t.commit ();
```

Note that as with all native views, the order and types of data members must match those of columns in the `SELECT` list inside the stored procedure.

There are also a number of limitations when it comes to support for MySQL stored procedures in ODB views. First of all, you have to use MySQL server and client libraries version 5.5.3 or later since this is the version in which support for calling stored procedures with prepared statements was first added (the `mysql_stmt_next_result()` function).

In MySQL, a stored procedure can produce multiple results. For example, if a stored procedure executes several `SELECT` statements, then the result of calling such a procedure consists of multiple row sets, one for each `SELECT` statement. Additionally, if the procedure has any `OUT` or `INOUT` parameters, then their values are returned as an additional special row set containing only a single row. Because such multiple row sets can contain varying number and type of columns, they cannot be all extracted into a single view. As a result, an ODB view will only extract the data from the first row set and ignore all the subsequent ones.

In particular, this means that we can use an ODB view to extract the values of the `OUT` and `INOUT` parameters provided that the stored procedure does not generate any other row sets. For example:

```
CREATE PROCEDURE person_min_max_age (
  OUT min_age SMALLINT,
  OUT max_age SMALLINT)
BEGIN
  SELECT MIN(age), MAX(age) INTO min_age, max_age FROM person;
END

#pragma db view query("CALL person_min_max_age((?))")
struct person_min_max_age
{
  unsigned short min_age;
  unsigned short max_age;
};

using query = odb::query<person_min_max_age>;

transaction t (db.begin ());

// We know this query always returns a single row, so use query_value().
// We have to pass dummy values for OUT parameters.
//
person_min_max_age mma (
  db.query_value<person_min_max_age> (
    query::_val (0) + "," + query::_val (0)));

cerr << mma.min_age << " " << mma.max_age << endl;

t.commit ();
```

Another limitation that stems from having multiple results is the inability to cache the result of a stored procedure call. In other words, a MySQL stored procedure call always produces an uncached query result (Section 4.4, "Query Result").

18 SQLite Database

To generate support code for the SQLite database you will need to pass the "`--database sqlite`" (or "`-d sqlite`") option to the ODB compiler. Your application will also need to link to the SQLite ODB runtime library (`libodb-sqlite`). All SQLite-specific ODB classes are defined in the `odb::sqlite` namespace.

18.1 SQLite Type Mapping

The following table summarizes the default mapping between basic C++ value types and SQLite database types. This mapping can be customized on the per-type and per-member basis using the ODB Pragma Language (Chapter 14, "ODB Pragma Language").

C++ Type	SQLite Type	Default `NULL` Semantics
`bool`	`INTEGER`	`NOT NULL`
`char`	`TEXT`	`NOT NULL`
`signed char`	`INTEGER`	`NOT NULL`
`unsigned char`	`INTEGER`	`NOT NULL`
`short`	`INTEGER`	`NOT NULL`
`unsigned short`	`INTEGER`	`NOT NULL`
`int`	`INTEGER`	`NOT NULL`
`unsigned int`	`INTEGER`	`NOT NULL`
`long`	`INTEGER`	`NOT NULL`
`unsigned long`	`INTEGER`	`NOT NULL`
`long long`	`INTEGER`	`NOT NULL`
`unsigned long long`	`INTEGER`	`NOT NULL`
`float`	`REAL`	`NULL`
`double`	`REAL`	`NULL`
`std::string`	`TEXT`	`NOT NULL`
`char[N]`	`TEXT`	`NOT NULL`
`std::wstring (Windows only)`	`TEXT`	`NOT NULL`
`wchar_t[N] (Windows only)`	`TEXT`	`NOT NULL`
`odb::sqlite::text`	`TEXT (STREAM)`	`NOT NULL`
`odb::sqlite::blob`	`BLOB (STREAM)`	`NOT NULL`

It is possible to map the `char` C++ type to the `INTEGER` SQLite type using the `db type` pragma (Section 14.4.3, "`type`").

SQLite represents the `NaN FLOAT` value as a `NULL` value. As a result, columns of the `float` and `double` types are by default declared as `NULL`. However, you can override this by explicitly declaring them as `NOT NULL` with the `db not_null` pragma (Section 14.4.6, "`null/not_null`").

Additionally, by default, C++ enums and C++11 enum classes are automatically mapped to the SQLite `INTEGER` type with the default `NULL` semantics being `NOT NULL`. For example:

```
enum color {red, green, blue};
enum class taste: unsigned char
{
  bitter = 1,
  sweet,
  sour = 4,
  salty
};

#pragma db object
class object
{
  ...

  color color_; // Automatically mapped to INTEGER.
  taste taste_; // Automatically mapped to INTEGER.
};
```

Note also that SQLite only operates with signed integers and the largest value that an SQLite database can store is a signed 64-bit integer. As a result, greater `unsigned long` and `unsigned long long` values will be represented in the database as negative values.

It is also possible to add support for additional SQLite types, such as `NUMERIC`. For more information, refer to Section 14.8, "Database Type Mapping Pragmas".

18.1.1 String Type Mapping

The SQLite ODB runtime library provides support for mapping the `std::array<char, N>` and, on Windows, `std::array<wchar_t, N>` types to the SQLite `TEXT` type. However, this mapping is not enabled by default (in particular, by default, `std::array` will be treated as a container). To enable the alternative mapping for this type we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("TEXT")
  std::array<char, 128> name_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using name_type = std::array<char, 128>;
#pragma db value(name_type) type("TEXT")

#pragma db object
class object
{
  ...

  name_type name_; // Mapped to TEXT.
};
```

The `char[N]`, `std::array<char, N>`, `wchar_t[N]`, and `std::array<wchar_t, N>` values may or may not be zero-terminated. When extracting such values from the database, ODB will append the zero terminator if there is enough space.

18.1.2 Binary Type Mapping

The SQLite ODB runtime library provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, `unsigned char[N]`, `std::array<char, N>`, and `std::array<unsigned char, N>` types to the SQLite `BLOB` type. However, these mappings are not enabled by default (in particular, by default, `std::vector` and `std::array` will be treated as containers). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("BLOB")
  unsigned char uuid_[16];
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using buffer = std::vector<char>;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Note also that in native queries (Chapter 4, "Querying the Database") `char[N]` and `std::array<char, N>` parameters are by default passed as a string rather than a binary. To pass such parameters as a binary, we need to specify the database type explicitly in the `_val()`/`_ref()` calls. Note also that we don't need to do this for the integrated queries, for example:

```
char u[16] = {...};

db.query<object> ("uuid = " + query::_val<odb::sqlite::id_blob> (u));
db.query<object> (query::uuid == query::_ref (u));
```

18.1.3 Incremental `BLOB`/`TEXT` I/O

This section describes the SQLite ODB runtime library support for incremental reading and writing of `BLOB` and `TEXT` values. The provided API is a thin wrapper around the native SQLite `sqlite3_blob_*()` function family. As a result, it is highly recommended that you familiarize yourself with the semantics of this SQLite functionality before continuing with this section.

The SQLite runtime provides the `blob` and `text` types that can be used to represent `BLOB` and `TEXT` data members that will be read/written using the incremental I/O. For example:

```
#include <odb/sqlite/blob.hxx>
#include <odb/sqlite/text.hxx>

#pragma db object
class object
{
public
  #pragma db id auto
  unsigned long long id;

  odb::sqlite::blob b; // Mapped to BLOB.
  odb::sqlite::text t; // Mapped to TEXT.
};
```

The `blob` and `text` types should be viewed as *descriptors* of the `BLOB` and `TEXT` values (rather than the values themselves) that can be used to *open* the values for reading or writing. These two types have an identical interface that is presented below. Notice that it is essentially the list of arguments (except for `size` which is discussed below) to the `sqlite3_blob_open()` function:

```
namespace odb
{
  namespace sqlite
  {
    class blob|text
    {
    public:
      explicit
      blob|text (std::size_t = 0);

      std::size_t size ()
      void        size (std::size_t);

      const std::string& db () const;
      const std::string& table () const;
      const std::string& column () const;
      long long          rowid () const;

      void
      clear ();
    };
  }
}
```

To read/write data from/to a incremental `BLOB` or `TEXT` value we use the corresponding `blob_stream` and `text_stream` stream types. Their interfaces closely mimic the underlying `sqlite3_blob_*()` functions and are presented below. Note that in order to create a stream we have to pass the corresponding descriptor:

```
#include <odb/sqlite/stream.hxx>

namespace odb
{
  namespace sqlite
  {
    class stream
    {
    public:
      stream (const char* db,
              const char* table,
              const char* column,
              long long rowid,
              bool rw);
```

```
      std::size_t
      size () const;

      // The following two functions throw std::invalid_argument if
      // offset + n is past size().
      //
      void
      read (void* buf, std::size_t n, std::size_t offset = 0);

      void
      write (const void* buf, std::size_t n, std::size_t offset = 0);

      sqlite3_blob*
      handle () const;

      // Close without reporting errors, if any.
      //
      ~stream ();

      // Close with reporting errors, if any.
      //
      void
      close ();

      // Open the same BLOB but in a different row. Can be faster
      // than creating a new stream instance. Note that the stream
      // must be in the open state prior to calling this function.
      //
      void
      reopen (long long rowid);
    };
  }
}

#include <odb/sqlite/blob-stream.hxx>

namespace odb
{
  namespace sqlite
  {
    class blob_stream: public stream
    {
    public:
      blob_stream (const blob&, bool rw);
    };
  }
}

#include <odb/sqlite/text-stream.hxx>
```

```
namespace odb
{
  namespace sqlite
  {
    class text_stream: public stream
    {
    public:
      text_stream (const text&, bool rw);
    };
  }
}
```

The `rw` argument to the constructors above specifies whether to open the value for reading only (`false`) or to read and write (`true`).

In SQLite the incremental `BLOB` and `TEXT` sizes are fixed in the sense that they must be specified before the object is persisted or updated and the following write operations can only write that much data. This is what the `size` data member in the descriptors is for. You can also determine the size of the opened value, for both reading and writing, using the `size()` stream function. The following example puts all of this together:

```
#include <odb/sqlite/blob-stream.hxx>
#include <odb/sqlite/text-stream.hxx>

string txt (1024 * 1024, 't');
vector<char> blb (1024 * 1024, 'b');

object o;

// Persist.
//
{
  transaction tx (db.begin ());

  // Specify the sizes of the values before calling persist().
  //
  o.t.size (txt.size ());
  o.b.size (blb.size ());

  db.persist (o);

  // Write the data.
  //
  blob_stream bs (o.b, true); // Open for read/write.
  assert (bs.size () == blb.size ());
  bs.write (blb.data (), blb.size ());

  text_stream ts (o.t, true); // Open for read/write.
  assert (ts.size () == txt.size ());
  ts.write (txt.data (), txt.size ());
```

```
  tx.commit ();
}

// Load.
//
{
  transaction tx (db.begin ());
  unique_ptr<object> p (db.load<object> (o.id));

  text_stream ts (p->t, false); // Open for reading.
  vector<char> t (ts.size () + 1, '\0');
  ts.read (t.data (), t.size () - 1);
  assert (string (t.data ()) == txt);

  blob_stream bs (p->b, false); // Open for reading.
  vector<char> b (bs.size (), '\0');
  bs.read (b.data (), b.size ());
  assert (b == blb);

  tx.commit ();
}

// Update
//
txt.resize (txt.size () + 1, 't');
txt[0] = 'A';
txt[txt.size () - 1] = 'Z';

blb.resize (blb.size () - 1);
blb.front () = 'A';
blb.back () = 'Z';

{
  transaction tx (db.begin ());

  // Specify the new sizes of the values before calling update().
  //
  o.t.size (txt.size ());
  o.b.size (blb.size ());

  db.update (o);

  // Write the data.
  //
  blob_stream bs (o.b, true);
  bs.write (blb.data (), blb.size ());

  text_stream ts (o.t, true);
```

```
  ts.write (txt.data (), txt.size ());

  tx.commit ();
}
```

For the most part, the incremental `BLOB` and `TEXT` descriptors can be used as any other simple values. Specifically, they can be used as container elements (Chapter 5, "Containers"), as `NULL`-able values (Section 7.3, "Pointers and NULL Value Semantics"), and in views (Chapter 10, "Views"). The following example illustrates the use within a view:

```
#pragma db view object(object)
struct load_b
{
  odb::sqlite::blob b;
};

using query = odb::query<load_b>;

transaction tx (db.begin ());

for (load_b& lb: db.query<load_b> (query::t == "test"))
{
  blob_stream bs (lb.b, false);
  vector<char> b (bs.size (), '\0');
  bs.read (b.data (), b.size ());
}

tx.commit ();
```

However, being a low-level, SQLite-specific mechanism, the incremental I/O has a number of nuances that should be kept in mind. Firstly, the streams should be opened within a transaction and, unless already closed, they will be automatically closed when the transaction is committed or rolled back. The following modification of the previous example helps to illustrate this point:

```
{
  transaction tx (db.begin ());

  // ...

  db.persist (o);

  blob_stream bs (o.b, true);

  tx.commit ();

  // ERROR: stream is closed.
  //
  bs.write (blb.data (), blb.size ());
}
```

```
// ERROR: not in transaction.
//
text_stream ts (o.t, true);
```

Because loading an object with an incremental `BLOB` or `TEXT` value involves additional actions after the database function returns (that is, reading the actual data), certain commonly-expected "round-trip" assumptions will no longer hold unless special steps are taken, for instance (again, continuing with our example):

```
transaction tx (db.begin ());

unique_ptr<object> p (db.load<object> (o.id));
p->name = "foo"; // Update some other member.
db.update (*p);  // Bad behavior: incremental BLOB/TEXT invalidated.

tx.commit ();
```

One way to restore the expected behavior is to place the incremental `BLOB` and `TEXT` values into their own, separately loaded/updated sections (Chapter 9, "Sections"). The alternative approach would be to perform the incremental I/O as part of the database operation `post_*` callbacks (Section 14.1.7, "`callback`").

Finally, note that when using incremental `TEXT` values, the data that we read/write is the raw bytes in the encoding used by the database (`UTF-8` by default; see SQLite `PRAGMA encoding` documentation for details).

18.1.4 Mixed Automatic/Manual Object Id Assignment

In SQLite an automatic object id can also be assigned manually. For example:

```
#pragma db id auto
odb::nullable<int64_t> id;
```

Then set the id member to `NULL` to get auto-assignment or to the actual value to use a manual id. This functionality is normally used to reserve a special id, typically `0`, for a special object.

18.2 SQLite Database Class

The SQLite `database` class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class database: public odb::database
    {
```

```
    public:
      database (const std::string& name,
                int flags = SQLITE_OPEN_READWRITE,
                bool foreign_keys = true,
                const std::string& vfs = "",
                std::[auto|unique]_ptr<connection_factory> = 0);

#ifdef _WIN32
      database (const std::wstring& name,
                int flags = SQLITE_OPEN_READWRITE,
                bool foreign_keys = true,
                const std::string& vfs = "",
                std::[auto|unique]_ptr<connection_factory> = 0);
#endif

      database (int& argc,
                char* argv[],
                bool erase = false,
                int flags = SQLITE_OPEN_READWRITE,
                bool foreign_keys = true,
                const std::string& vfs = "",
                std::[auto|unique]_ptr<connection_factory> = 0);

      static void
      print_usage (std::ostream&);

    public:
      const std::string&
      name () const;

      int
      flags () const;

    public:
      transaction
      begin_immediate ();

      transaction
      begin_exclusive ();

    public:
      connection_ptr
      connection ();
    };
  }
}
```

You will need to include the `<odb/sqlite/database.hxx>` header file to make this class available in your application.

The first constructor opens the specified SQLite database. The `name` argument is the database file name to open in the UTF-8 encoding. If this argument is empty, then a temporary, on-disk database is created. If this argument is the `:memory:` special value, then a temporary, in-memory database is created. The `flags` argument allows us to specify SQLite opening flags. For more information on the possible values, refer to the `sqlite3_open_v2()` function description in the SQLite C API documentation. The `foreign_keys` argument specifies whether foreign key constraints checking should be enabled. See Section 18.6.3, "Foreign Key Constraints" for more information on foreign keys. The `vfs` argument specifies the SQLite virtual file system module that should be used to access the database. If this argument is empty, then the default vfs module is used. Again, refer to the `sqlite3_open_v2()` function documentation for detail.

The following example shows how we can open the `test.db` database in the read-write mode and create it if it does not exist:

```
unique_ptr<odb::database> db (
  new odb::sqlite::database (
    "test.db",
    SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE));
```

The second constructor is the same as the first except that the database name is passes as `std::wstring` in the UTF-16 encoding. This constructor is only available when compiling for Windows.

The third constructor extracts the database parameters from the command line. The following options are recognized:

```
--database <name>
--create
--read-only
--options-file <file>
```

By default, this constructor opens the database in the read-write mode (`SQLITE_OPEN_READWRITE` flag). If the `--create` flag is specified, then the database file is created if it does not already exist (`SQLITE_OPEN_CREATE` flag). If the `--read-only` flag is specified, then the database is opened in the read-only mode (`SQLITE_OPEN_READONLY` flag instead of `SQLITE_OPEN_READWRITE`). The `--options-file` option allows us to specify some or all of the database options in a file with each option appearing on a separate line followed by a space and an option value.

If the `erase` argument to this constructor is true, then the above options are removed from the `argv` array and the `argc` count is updated accordingly. This is primarily useful if your application accepts other options or arguments and you would like to get the SQLite options out of the `argv` array.

The `flags` argument has the same semantics as in the first constructor. Flags from the command line always override the corresponding values specified with this argument.

The third constructor throws the `odb::sqlite::cli_exception` exception if the SQLite option values are missing or invalid. See Section 18.5, "SQLite Exceptions" for more information on this exception.

The static `print_usage()` function prints the list of options with short descriptions that are recognized by the third constructor.

The last argument to all of the constructors is a pointer to the connection factory. In C++98/03, it is `std::auto_ptr` while in C++11 `std::unique_ptr` is used instead. If we pass a non-`NULL` value, the database instance assumes ownership of the factory instance. The connection factory interface as well as the available implementations are described in the next section.

The set of accessor functions following the constructors allows us to query the parameters of the `database` instance.

The `begin_immediate()` and `begin_exclusive()` functions are the SQLite-specific extensions to the standard `odb::database::begin()` function (see Section 3.5, "Transactions"). They allow us to start an immediate (`BEGIN IMMEDIATE`) and an exclusive (`BEGIN EXCLUSIVE`) SQLite transaction, respectively. For more information on the semantics of the immediate and exclusive transactions, refer to the `BEGIN` statement description in the SQLite documentation.

The `connection()` function returns a pointer to the SQLite database connection encapsulated by the `odb::sqlite::connection` class. For more information on `sqlite::connection`, refer to Section 18.3, "SQLite Connection and Connection Factory".

18.3 SQLite Connection and Connection Factory

The `sqlite::connection` class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class connection: public odb::connection
    {
    public:
      connection (database&, int extra_flags = 0);
      connection (database&, sqlite3*);

      transaction
      begin_immediate ();
```

```
      transaction
      begin_exclusive ();

      sqlite3*
      handle ();
    };

    using connection_ptr = details::shared_ptr<connection>;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.6, "Connections". The first overloaded `sqlite::connection` constructor opens a new SQLite connection. The `extra_flags` argument can be used to specify extra `sqlite3_open_v2()` flags that are combined with the flags specified in the `sqlite::database` constructor. The second constructor allows us to create a `connection` instance by providing an already open native SQLite handle. Note that the `connection` instance assumes ownership of this handle.

The `begin_immediate()` and `begin_exclusive()` functions allow us to start an immediate and an exclusive SQLite transaction on the connection, respectively. Their semantics are equivalent to the corresponding functions defined in the `sqlite::database` class (Section 18.2, "SQLite Database Class"). The `handle()` accessor returns the SQLite handle corresponding to the connection.

The `sqlite::connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class connection_factory
    {
    public:
      virtual void
      database (database&) = 0;

      virtual connection_ptr
      connect () = 0;
    };
  }
}
```

The `database()` function is called when a connection factory is associated with a database instance. This happens in the `odb::sqlite::database` class constructors. The `connect()` function is called whenever a database connection is requested.

The four implementations of the `connection_factory` interface provided by the SQLite ODB runtime library are `serial_connection_factory`, `single_connection_factory`, `new_connection_factory`, and `connection_pool_factory`. You will need to include the `<odb/sqlite/connection-factory.hxx>` header file to make the `connection_factory` interface and these implementation classes available in your application.

The `serial_connection_factory` class creates a single connection that is expected to be used by an application in a serialized manner. For example, a single-threaded application that executes all the database operations via the database instance and without dealing with multiple connections/transactions would qualify. The `serial_connection_factory` class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class serial_connection_factory: public connection_factory
    {
    public:
      serial_connection_factory ();

    protected:
      virtual connection_ptr
      create ();
    };
};
```

The `create()` virtual function is called when the factory needs to create the connection. By deriving from the `serial_connection_factory` class and overriding this function we can implement custom connection establishment and configuration.

The `single_connection_factory` class creates a single connection that is shared between all the threads in an application. If the connection is currently not in use, then it is returned to the caller. Otherwise, the caller is blocked until the connection becomes available. The `single_connection_factory` class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class single_connection_factory: public connection_factory
    {
    public:
      single_connection_factory ();

    protected:
      class single_connection: public connection
```

```
      {
      public:
        single_connection (database&, int extra_flags = 0);
        single_connection (database&, sqlite3*);
      };

      using single_connection_ptr = details::shared_ptr<single_connection>;

      virtual single_connection_ptr
      create ();
    };
};
```

The `create()` virtual function is called when the factory needs to create the connection. By deriving from the `single_connection_factory` class and overriding this function we can implement custom connection establishment and configuration.

The `new_connection_factory` class creates a new connection whenever one is requested. When a connection is no longer needed, it is released and closed. The `new_connection_factory` class has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class new_connection_factory: public connection_factory
    {
    public:
      new_connection_factory ();
    };
};
```

The `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace sqlite
  {
    class connection_pool_factory: public connection_factory
    {
    public:
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0);

    protected:
      class pooled_connection: public connection
      {
      public:
        pooled_connection (database_type&, int extra_flags = 0);
```

```
        pooled_connection (database_type&, sqlite3*);
      };

      using pooled_connection_ptr = details::shared_ptr<pooled_connection>;

      virtual pooled_connection_ptr
      create ();
    };
};
```

The `max_connections` argument in the `connection_pool_factory` constructor specifies the maximum number of concurrent connections that this pool factory will maintain. Similarly, the `min_connections` argument specifies the minimum number of available connections that should be kept open.

Whenever a connection is requested, the pool factory first checks if there is an unused connection that can be returned. If there is none, the pool factory checks the `max_connections` value to see if a new connection can be created. If the total number of connections maintained by the pool is less than this value, then a new connection is created and returned. Otherwise, the caller is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked callers waiting for a connection. If so, then one of them is unblocked and is given the connection. Otherwise, the pool factory checks whether the total number of connections maintained by the pool is greater than the `min_connections` value. If that's the case, the connection is closed. Otherwise, the connection is added to the pool of available connections to be returned on the next request. In other words, if the number of connections maintained by the pool exceeds `min_connections` and there are no callers waiting for a new connection, then the pool will close the excess connections.

If the `max_connections` value is 0, then the pool will create a new connection whenever all of the existing connections are in use. If the `min_connections` value is 0, then the pool will never close a connection and instead maintain all the connections that were ever created.

The `create()` virtual function is called whenever the pool needs to create a new connection. By deriving from the `connection_pool_factory` class and overriding this function we can implement custom connection establishment and configuration.

By default, connections created by `new_connection_factory` and `connection_pool_factory` enable the SQLite shared cache mode and use the unlock notify functionality to aid concurrency. To disable the shared cache mode you can pass the `SQLITE_OPEN_PRIVATECACHE` flag when creating the database instance. For more information on the shared cache mode refer to the SQLite documentation.

If you pass `NULL` as the connection factory to one of the `database` constructors, then the `connection_pool_factory` instance will be created by default with the min and max connections values set to `0`. The following code fragment shows how we can pass our own connection factory instance:

```
#include <odb/database.hxx>

#include <odb/sqlite/database.hxx>
#include <odb/sqlite/connection-factory.hxx>

int
main (int argc, char* argv[])
{
  unique_ptr<odb::sqlite::connection_factory> f (
    new odb::sqlite::connection_pool_factory (20));

  unique_ptr<odb::database> db (
    new sqlite::database (argc, argv, false, SQLITE_OPEN_READWRITE, f));
}
```

18.4 Attached SQLite Databases

The SQLite database class provides support for attaching additional databases to the main database connections using the SQLite `ATTACH DATABASE` statement. Good understanding of the SQLite attached database semantics and ODB connection management is strongly recommended when using this functionality.

The relevant part of the SQLite database class interface is shown below:

```
namespace odb
{
  namespace sqlite
  {
    class database: public odb::database
    {
    public:

      ...

      database (const connection_ptr&,
                const std::string& name,
                const std::string& schema,
                std::[auto|unique]_ptr<attached_connection_factory> = 0);

      void
      detach ();

      database&
      main_database ();
```

```
      const std::string&
      schema () const;
    };
  }
}
```

The shown constructor attaches to the specified connection a database with the specified name as the specified schema.

The resulting database instance is referred to as an *attached database* and the connection it returns as an *attached connection* (which is just a proxy for the main connection). Database operations executed on the attached database or attached connection are automatically translated to refer to the specified schema rather than `"main"`. For uniformity attached databases can also be created for the pre-attached `"main"` and `"temp"` schemas (in this case the `name` argument can be anything).

The automatic translation of the statements relies on their text having references to top-level database entities (tables, indexes, etc) qualified with the `"main"` schema. To achieve this, compile your headers with the `--schema main` option and, if using schema migration, with the `--schema-version-table main.schema_version` option. You must also not use `"main"` as an object/table alias in views of native statements. For optimal translation performance use 4-character schema names.

The main connection and attached to it databases and connections are all meant to be used within the same thread. In particular, the attached database holds a counted reference to the main connection which means the connection will not be released until all the attached to this connection databases are destroyed.

Note that in this model the attached databases are attached to the main connection, not to the (main) database, which mimics the underlying semantics of SQLite. An alternative model would have been to notionally attach the databases to the main database and under the hood automatically attach them to each returned connection. While this may seem like a more convenient model in some cases, it is also less flexible: the current model allows attaching a different set of databases to different connections, attaching them on demand as the transaction progresses, etc. Also, the more convenient model can be implemented on top of this model by deriving an application-specific database class and/or providing custom connection factories.

Note also that unless the name is a URI with appropriate mode, the attached database is opened with the `SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE` flags. In particular, if you want just `SQLITE_OPEN_READWRITE`, then you will need to verify the database file existence manually prior to calling this constructor.

Note that attaching/detaching databases within a transaction is only supported since SQLite 3.21.0.

The `detach()` function detaches the attached database. The database is automatically detached on destruction but a failure to detach is ignored. To detect such a failure perform explicit detach. For uniformity detaching a main database is a no-op.

The `main_database()` function returns the main database of an attached database. If this database is main, return itself.

The `schema()` function returns the schema name under which this database was attached or empty if this is the main database.

18.5 SQLite Exceptions

The SQLite ODB runtime library defines the following SQLite-specific exceptions:

```
namespace odb
{
  namespace sqlite
  {
    class forced_rollback: odb::recoverable
    {
    public:
      virtual const char*
      what () const throw ();
    };

    class database_exception: odb::database_exception
    {
    public:
      int
      error () const

      int
      extended_error () const;

      const std::string&
      message () const;

      virtual const char*
      what () const throw ();
    };

    class cli_exception: odb::exception
    {
    public:
      virtual const char*
```

```
      what () const throw ();
    };
  }
}
```

You will need to include the `<odb/sqlite/exceptions.hxx>` header file to make these exceptions available in your application.

The `odb::sqlite::forced_rollback` exception is thrown if SQLite is forcing the current transaction to roll back. For more information on this behavior refer to Section 18.6.6, "Forced Rollback".

The `odb::sqlite::database_exception` is thrown if an SQLite database operation fails. The SQLite-specific error information is accessible via the `error()`, `extended_error()`, and `message()` functions. All this information is also combined and returned in a human-readable form by the `what()` function.

The `odb::sqlite::cli_exception` is thrown by the command line parsing constructor of the `odb::sqlite::database` class if the SQLite option values are missing or invalid. The `what()` function returns a human-readable description of an error.

18.6 SQLite Limitations

The following sections describe SQLite-specific limitations imposed by the current SQLite and ODB runtime versions.

18.6.1 Query Result Caching

SQLite ODB runtime implementation does not perform query result caching (Section 4.4, "Query Result") even when explicitly requested. The SQLite API supports interleaving execution of multiple prepared statements on a single connection. As a result, with SQLite, it is possible to have multiple uncached results and calls to other database functions do not invalidate them. The only limitation of the uncached SQLite results is the unavailability of the `result::size()` function. If you call this function on an SQLite query result, then the `odb::result_not_cached` exception (Section 3.14, "ODB Exceptions") is always thrown. Future versions of the SQLite ODB runtime library may add support for result caching.

18.6.2 Automatic Assignment of Object Ids

Due to SQLite API limitations, every automatically assigned object id (Section 14.4.2, "`auto`") should have the `INTEGER` SQLite type. While SQLite will treat other integer type names (such as `INT`, `BIGINT`, etc.) as `INTEGER`, automatic id assignment will not work. By default, ODB maps all C++ integral types to `INTEGER`. This means that the only situation that requires consideration is the assignment of a custom database type using the `db type` pragma (Section 14.4.3,

"type"). For example:

```
#pragma db object
class person
{
  ...

  //#pragma db id auto type("INT")     // Will not work.
  //#pragma db id auto type("INTEGER") // Ok.
  #pragma db id auto                   // Ok, Mapped to INTEGER.
  unsigned int id_;
};
```

18.6.3 Foreign Key Constraints

By default the SQLite ODB runtime enables foreign key constraints checking (PRAGMA foreign_keys=ON). You can disable foreign keys by passing false as the foreign_keys argument to one of the odb::sqlite::database constructors. Foreign keys will also be disabled if the SQLite library is built without support for foreign keys (SQLITE_OMIT_FOREIGN_KEY and SQLITE_OMIT_TRIGGER macros) or if you are using an SQLite version prior to 3.6.19, which does not support foreign key constraints checking.

If foreign key constraints checking is disabled or not available, then inconsistencies in object relationships will not be detected. Furthermore, using the erase_query() function (Section 3.11, "Deleting Persistent Objects") to delete persistent objects that contain containers will not work correctly. Container data for such objects will not be deleted.

When foreign key constraints checking is enabled, then you may get the "foreign key constraint failed" error while re-creating the database schema. This error is due to bugs in the SQLite DDL foreign keys support. The recommended work-around for this problem is to temporarily disable foreign key constraints checking while re-creating the schema. The following code fragment shows how this can be done:

```
#include <odb/connection.hxx>
#include <odb/transaction.hxx>
#include <odb/schema-catalog.hxx>

odb::database& db = ...

{
  odb::connection_ptr c (db.connection ());

  c->execute ("PRAGMA foreign_keys=OFF");

  odb::transaction t (c->begin ());
  odb::schema_catalog::create_schema (db);
```

```
  t.commit ();

  c->execute ("PRAGMA foreign_keys=ON");
}
```

Finally, ODB assumes the standard SQL behavior which requires that foreign key constraints checking is deferred until the transaction is committed. Default SQLite behavior is to check such constraints immediately. As a result, when used with ODB, a custom database schema that defines foreign key constraints may need to declare such constraints as `DEFERRABLE INITIALLY DEFERRED`, as shown in the following example. By default, schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer INTEGER REFERENCES Employer(id)
           DEFERRABLE INITIALLY DEFERRED);
```

You can override the default behavior and instruct the ODB compiler to generate non-deferrable foreign keys by specifying the `--fkeys-deferrable-mode not_deferrable` ODB compiler option. Note, however, that in this case the order in which you persist, update, and erase objects within a transaction becomes important.

18.6.4 Constraint Violations

Due to the granularity of the SQLite error codes, it is impossible to distinguish between the duplicate primary key and other constraint violations. As a result, when making an object persistent, the SQLite ODB runtime will translate all constraint violation errors to the `object_already_persistent` exception (Section 3.14, "ODB Exceptions").

18.6.5 Sharing of Queries

As discussed in Section 4.3, "Executing a Query", a query instance that does not have any by-reference parameters is immutable and can be shared between multiple threads without synchronization. Currently, the SQLite ODB runtime does not support this functionality. Future versions of the library will remove this limitation.

18.6.6 Forced Rollback

In SQLite 3.7.11 or later, if one of the connections participating in the shared cache rolls back a transaction, then ongoing transactions on other connections in the shared cache may also be forced to roll back. An example of such behavior would be a read-only transaction that is forced to roll back while iterating over the query result because another transaction on another connection was rolled back.

If a transaction is being forced to roll back by SQLite, then ODB throws `odb::sqlite::forced_rollback` (Section 18.5, "SQLite Exceptions") which is a recoverable exception (3.7 Error Handling and Recovery). As a result, the recommended way to handle this exception is to re-execute the affected transaction.

18.6.7 Database Schema Evolution

From the list of schema migration changes supported by ODB (Section 13.2, "Schema Migration"), the following are not supported by SQLite:

- drop column
- alter column, set `NULL`/`NOT NULL`
- add foreign key
- drop foreign key

The biggest problem is the lack of support for dropping columns. This means that it would be impossible to delete a data member in a persistent class. To work around this limitation ODB implements *logical delete* for columns that allow `NULL` values. In this case, instead of dropping the column (in the post-migration stage), the schema migration statements will automatically reset this column in all the existing rows to `NULL`. Any new rows that are inserted later will also automatically have this column set to `NULL` (unless the column specifies a default value).

Since it is also impossible to change the column's `NULL`/`NOT NULL` attribute after it has been added, to make schema evolution support usable in SQLite, all the columns should be added as `NULL` even if semantically they should not allow `NULL` values. We should also normally refrain from assigning default values to columns (Section 14.4.7, `default`), unless the space overhead of a default value is not a concern. Explicitly making all the data members `NULL` would be burdensome and ODB provides the `--sqlite-override-null` command line option that forces all the columns, even those that were explicitly marked `NOT NULL`, to be `NULL` in SQLite.

SQLite only supports adding foreign keys as part of the column addition. As a result, we can only add a new data member of an object pointer type if it points to an object with a simple (single-column) object id.

SQLite also doesn't support dropping foreign keys. Leaving a foreign key around works well with logical delete unless we also want to delete the pointed-to object. In this case we will have to leave an empty table corresponding to the pointed-to object around. An alternative would be to make a copy of the pointing object without the object pointer, migrate the data, and then delete both the old pointing and the pointed-to objects. Since this will result in dropping the pointing table, the foreign key will be dropped as well. Yet another, more radical, solution to this problem is to disable foreign keys checking altogether (see the `foreign_keys` SQLite pragma).

To summarize, to make schema evolution support usable in SQLite we should pass the `--sqlite-override-null` option when compiling our persistent classes and also refrain from assigning default values to data members. Note also that this has to be done from the start so that every column is added as `NULL` and therefore can be logically deleted later. In particular, you cannot add the `--sqlite-override-null` option when you realize you need to delete a data member. At this point it is too late since the column has already been added as `NOT NULL` in existing databases. We should also avoid composite object ids if we are planning to use object relationships.

18.7 SQLite Index Definitions

When the `index` pragma (Section 14.7, "Index Definition Pragmas") is used to define an SQLite index, the `type` clause specifies the index type (for example, `UNIQUE`) while the `method` and `options` clauses are not used. The column options can be used to specify collations and the sort order. For example:

```
#pragma db object
class object
{
  ...

  std::string name_;

  #pragma db index member(name_, "COLLATE binary DESC")
};
```

Index names in SQLite are database-global. To avoid name clashes, ODB automatically prefixes each index name with the table name on which it is defined.

19 PostgreSQL Database

To generate support code for the PostgreSQL database you will need to pass the "`--database pgsql`" (or "`-d pgsql`") option to the ODB compiler. Your application will also need to link to the PostgreSQL ODB runtime library (`libodb-pgsql`). All PostgreSQL-specific ODB classes are defined in the `odb::pgsql` namespace.

ODB utilizes prepared statements extensively. Support for prepared statements was added in PostgreSQL version 7.4 with the introduction of the messaging protocol version 3.0. For this reason, ODB supports only PostgreSQL version 7.4 and later.

19.1 PostgreSQL Type Mapping

The following table summarizes the default mapping between basic C++ value types and PostgreSQL database types. This mapping can be customized on the per-type and per-member basis using the ODB Pragma Language (Chapter 14, "ODB Pragma Language").

C++ Type	PostgreSQL Type	Default `NULL` Semantics
`bool`	`BOOLEAN`	`NOT NULL`
`char`	`CHAR(1)`	`NOT NULL`
`signed char`	`SMALLINT`	`NOT NULL`
`unsigned char`	`SMALLINT`	`NOT NULL`
`short`	`SMALLINT`	`NOT NULL`
`unsigned short`	`SMALLINT`	`NOT NULL`
`int`	`INTEGER`	`NOT NULL`
`unsigned int`	`INTEGER`	`NOT NULL`
`long`	`BIGINT`	`NOT NULL`
`unsigned long`	`BIGINT`	`NOT NULL`
`long long`	`BIGINT`	`NOT NULL`
`unsigned long long`	`BIGINT`	`NOT NULL`
`float`	`REAL`	`NOT NULL`
`double`	`DOUBLE PRECISION`	`NOT NULL`
`std::string`	`TEXT`	`NOT NULL`
`char[N]`	`VARCHAR(N-1)`	`NOT NULL`

It is possible to map the `char` C++ type to an integer database type (for example, `SMALLINT`) using the `db type` pragma (Section 14.4.3, "`type`").

Additionally, by default, C++ enums and C++11 enum classes are automatically mapped to the PostgreSQL types corresponding to their underlying integral types (see table above). The default `NULL` semantics is `NOT NULL`. For example:

```
enum color {red, green, blue};
enum class taste: unsigned char
{
  bitter = 1,
  sweet,
  sour = 4,
  salty
};

#pragma db object
```

```
class object
{
  ...

  color color_; // Automatically mapped to INTEGER.
  taste taste_; // Automatically mapped to SMALLINT.
};
```

Note also that because PostgreSQL does not support unsigned integers, the `unsigned short`, `unsigned int`, and `unsigned long`/`unsigned long long` C++ types are by default mapped to the `SMALLINT`, `INTEGER`, and `BIGINT` PostgreSQL types, respectively. The sign bit of the value stored by the database for these types will contain the most significant bit of the actual unsigned value being persisted.

It is also possible to add support for additional PostgreSQL types, such as `NUMERIC`, geometry types, `XML`, `JSON`, enumeration types, composite types, arrays, geospatial types, and the key-value store (`HSTORE`). For more information, refer to Section 14.8, "Database Type Mapping Pragmas".

19.1.1 String Type Mapping

The PostgreSQL ODB runtime library provides support for mapping the `std::string`, `char[N]`, and `std::array<char, N>` types to the PostgreSQL `CHAR`, `VARCHAR`, and `TEXT` types. However, these mappings are not enabled by default (in particular, by default, `std::array` will be treated as a container). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("CHAR(2)")
  char state_[2];

  #pragma db type("VARCHAR(128)")
  std::string name_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
#pragma db value(std::string) type("VARCHAR(128)")

#pragma db object
class object
{
  ...

  std::string name_; // Mapped to VARCHAR(128).
};
```

The `char[N]` and `std::array<char, N>` values may or may not be zero-terminated. When extracting such values from the database, ODB will append the zero terminator if there is enough space.

19.1.2 Binary Type and `UUID` Mapping

The PostgreSQL ODB runtime library provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, `unsigned char[N]`, `std::array<char, N>`, and `std::array<unsigned char, N>` types to the PostgreSQL `BYTEA` type. There is also support for mapping the `char[16]` array to the PostgreSQL `UUID` type. However, these mappings are not enabled by default (in particular, by default, `std::vector` and `std::array` will be treated as containers). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type("UUID")
  char uuid_[16];

  #pragma db type("BYTEA")
  std::vector<char> buf_;

  #pragma db type("BYTEA")
  unsigned char data_[256];
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using buffer = std::vector<char>;
#pragma db value(buffer) type("BYTEA")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BYTEA.
};
```

Note also that in native queries (Chapter 4, "Querying the Database") `char[N]` and `std::array<char, N>` parameters are by default passed as a string rather than a binary. To pass such parameters as a binary, we need to specify the database type explicitly in the `_val()`/`_ref()` calls. Note also that we don't need to do this for the integrated queries, for example:

```
char u[16] = {...};

db.query<object> ("uuid = " + query::_val<odb::pgsql::id_uuid> (u));
db.query<object> ("buf = " + query::_val<odb::pgsql::id_bytea> (u));
db.query<object> (query::uuid == query::_ref (u));
```

19.2 PostgreSQL Database Class

The PostgreSQL `database` class has the following interface:

```
namespace odb
{
  namespace pgsql
  {
    class database: public odb::database
    {
    public:
      database (const std::string& user,
                const std::string& password,
                const std::string& db,
                const std::string& host = "",
                unsigned int port = 0,
                const std::string& extra_conninfo = "",
                std::[auto|unique]_ptr<connection_factory> = 0);

      database (const std::string& user,
                const std::string& password,
                const std::string& db,
                const std::string& host,
                const std::string& socket_ext,
                const std::string& extra_conninfo = "",
                std::[auto|unique]_ptr<connection_factory> = 0);
```

```
      database (const std::string& conninfo,
                std::[auto|unique]_ptr<connection_factory> = 0);

      database (int& argc,
                char* argv[],
                bool erase = false,
                const std::string& extra_conninfo = "",
                std::[auto|unique]_ptr<connection_factory> = 0);

      static void
      print_usage (std::ostream&);

    public:
      const std::string&
      user () const;

      const std::string&
      password () const;

      const std::string&
      db () const;

      const std::string&
      host () const;

      unsigned int
      port () const;

      const std::string&
      socket_ext () const;

      const std::string&
      extra_conninfo () const;

      const std::string&
      conninfo () const;

    public:
      connection_ptr
      connection ();
    };
  }
}
```

You will need to include the `<odb/pgsql/database.hxx>` header file to make this class available in your application.

The overloaded `database` constructors allow us to specify the PostgreSQL database parameters that should be used when connecting to the database. The `port` argument in the first constructor is an integer value specifying the TCP/IP port number to connect to. A zero port number indicates that the default port should be used. The `socket_ext` argument in the second constructor is a string value specifying the UNIX-domain socket file name extension.

The third constructor allows us to specify all the database parameters as a single `conninfo` string. All other constructors accept additional database connection parameters as the `extra_conninfo` argument. For more information about the format of the `conninfo` string, refer to the `PQconnectdb()` function description in the PostgreSQL documentation. In the case of `extra_conninfo`, all the database parameters provided in this string will take precedence over those explicitly specified with other constructor arguments.

The last constructor extracts the database parameters from the command line. The following options are recognized:

```
--user <login> | --username <login>
--password <password>
--database <name> | --dbname <name>
--host <host>
--port <integer>
--options-file <file>
```

The `--options-file` option allows us to specify some or all of the database options in a file with each option appearing on a separate line followed by a space and an option value.

If the `erase` argument to this constructor is true, then the above options are removed from the `argv` array and the `argc` count is updated accordingly. This is primarily useful if your application accepts other options or arguments and you would like to get the PostgreSQL options out of the `argv` array.

This constructor throws the `odb::pgsql::cli_exception` exception if the PostgreSQL option values are missing or invalid. See section Section 19.4, "PostgreSQL Exceptions" for more information on this exception.

The static `print_usage()` function prints the list of options with short descriptions that are recognized by this constructor.

The last argument to all of the constructors is a pointer to the connection factory. In C++98/03, it is `std::auto_ptr` while in C++11 `std::unique_ptr` is used instead. If we pass a non-`NULL` value, the database instance assumes ownership of the factory instance. The connection factory interface as well as the available implementations are described in the next section.

The set of accessor functions following the constructors allows us to query the parameters of the `database` instance. Note that the `conninfo()` accessor returns a complete `conninfo` string which includes parameters that were explicitly specified with the various constructor arguments, as well as the extra parameters passed in the `extra_conninfo` argument. The `extra_conninfo()` accessor will return the `conninfo` string as passed in the `extra_conninfo` argument.

The `connection()` function returns a pointer to the PostgreSQL database connection encapsulated by the `odb::pgsql::connection` class. For more information on `pgsql::connection`, refer to Section 19.3, "PostgreSQL Connection and Connection Factory".

19.3 PostgreSQL Connection and Connection Factory

The `pgsql::connection` class has the following interface:

```
namespace odb
{
  namespace pgsql
  {
    class connection: public odb::connection
    {
    public:
      connection (database&);
      connection (database&, PGconn*);

      PGconn*
      handle ();
    };

    using connection_ptr = details::shared_ptr<connection>;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.6, "Connections". The first overloaded `pgsql::connection` constructor establishes a new PostgreSQL connection. The second constructor allows us to create a `connection` instance by providing an already connected native PostgreSQL handle. Note that the `connection` instance assumes ownership of this handle. The `handle()` accessor returns the PostgreSQL handle corresponding to the connection.

The `pgsql::connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace pgsql
  {
```

```
    class connection_factory
    {
    public:
      virtual void
      database (database&) = 0;

      virtual connection_ptr
      connect () = 0;
    };
  }
}
```

The `database()` function is called when a connection factory is associated with a database instance. This happens in the `odb::pgsql::database` class constructors. The `connect()` function is called whenever a database connection is requested.

The two implementations of the `connection_factory` interface provided by the PostgreSQL ODB runtime are `new_connection_factory` and `connection_pool_factory`. You will need to include the `<odb/pgsql/connection-factory.hxx>` header file to make the `connection_factory` interface and these implementation classes available in your application.

The `new_connection_factory` class creates a new connection whenever one is requested. When a connection is no longer needed, it is released and closed. The `new_connection_factory` class has the following interface:

```
namespace odb
{
  namespace pgsql
  {
    class new_connection_factory: public connection_factory
    {
    public:
      new_connection_factory ();
    };
};
```

The `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace pgsql
  {
    class connection_pool_factory: public connection_factory
    {
    public:
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0);
```

```
    protected:
      class pooled_connection: public connection
      {
      public:
        pooled_connection (database_type&);
        pooled_connection (database_type&, PGconn*);
      };

      using pooled_connection_ptr = details::shared_ptr<pooled_connection>;

      virtual pooled_connection_ptr
      create ();
    };
};
```

The `max_connections` argument in the `connection_pool_factory` constructor specifies the maximum number of concurrent connections that this pool factory will maintain. Similarly, the `min_connections` argument specifies the minimum number of available connections that should be kept open.

Whenever a connection is requested, the pool factory first checks if there is an unused connection that can be returned. If there is none, the pool factory checks the `max_connections` value to see if a new connection can be created. If the total number of connections maintained by the pool is less than this value, then a new connection is created and returned. Otherwise, the caller is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked callers waiting for a connection. If so, then one of them is unblocked and is given the connection. Otherwise, the pool factory checks whether the total number of connections maintained by the pool is greater than the `min_connections` value. If that's the case, the connection is closed. Otherwise, the connection is added to the pool of available connections to be returned on the next request. In other words, if the number of connections maintained by the pool exceeds `min_connections` and there are no callers waiting for a new connection, the pool will close the excess connections.

If the `max_connections` value is 0, then the pool will create a new connection whenever all of the existing connections are in use. If the `min_connections` value is 0, then the pool will never close a connection and instead maintain all the connections that were ever created.

The `create()` virtual function is called whenever the pool needs to create a new connection. By deriving from the `connection_pool_factory` class and overriding this function we can implement custom connection establishment and configuration.

If you pass `NULL` as the connection factory to one of the `database` constructors, then the `connection_pool_factory` instance will be created by default with the min and max connections values set to `0`. The following code fragment shows how we can pass our own

connection factory instance:

```
#include <odb/database.hxx>

#include <odb/pgsql/database.hxx>
#include <odb/pgsql/connection-factory.hxx>

int
main (int argc, char* argv[])
{
  unique_ptr<odb::pgsql::connection_factory> f (
    new odb::pgsql::connection_pool_factory (20));

  unique_ptr<odb::database> db (
    new pgsql::database (argc, argv, false, "", f));
}
```

19.4 PostgreSQL Exceptions

The PostgreSQL ODB runtime library defines the following PostgreSQL-specific exceptions:

```
namespace odb
{
  namespace pgsql
  {
    class database_exception: odb::database_exception
    {
    public:
      const std::string&
      message () const;

      const std::string&
      sqlstate () const;

      virtual const char*
      what () const throw ();
    };

    class cli_exception: odb::exception
    {
    public:
      virtual const char*
      what () const throw ();
    };
  }
}
```

You will need to include the `<odb/pgsql/exceptions.hxx>` header file to make these exceptions available in your application.

The `odb::pgsql::database_exception` is thrown if a PostgreSQL database operation fails. The PostgreSQL-specific error information is accessible via the `message()` and `sqlstate()` functions. All this information is also combined and returned in a human-readable form by the `what()` function.

The `odb::pgsql::cli_exception` is thrown by the command line parsing constructor of the `odb::pgsql::database` class if the PostgreSQL option values are missing or invalid. The `what()` function returns a human-readable description of an error.

19.5 PostgreSQL Limitations

The following sections describe PostgreSQL-specific limitations imposed by the current PostgreSQL and ODB runtime versions.

19.5.1 Query Result Caching

The PostgreSQL ODB runtime implementation will always return a cached query result (Section 4.4, "Query Result") even when explicitly requested not to. This is a limitation of the PostgreSQL client library (`libpq`) which does not support uncached (streaming) query results.

19.5.2 Foreign Key Constraints

ODB assumes the standard SQL behavior which requires that foreign key constraints checking is deferred until the transaction is committed. Default PostgreSQL behavior is to check such constraints immediately. As a result, when used with ODB, a custom database schema that defines foreign key constraints may need to declare such constraints as `INITIALLY DEFERRED`, as shown in the following example. By default, schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer BIGINT REFERENCES Employer(id) INITIALLY DEFERRED);
```

You can override the default behavior and instruct the ODB compiler to generate non-deferrable foreign keys by specifying the `--fkeys-deferrable-mode not_deferrable` ODB compiler option. Note, however, that in this case the order in which you persist, update, and erase objects within a transaction becomes important.

19.5.3 Unique Constraint Violations

Due to the granularity of the PostgreSQL error codes, it is impossible to distinguish between the duplicate primary key and other unique constraint violations. As a result, when making an object persistent, the PostgreSQL ODB runtime will translate all unique constraint violation errors to the `object_already_persistent` exception (Section 3.14, "ODB Exceptions").

19.5.4 Date-Time Format

ODB expects the PostgreSQL server to use integers as a binary format for the date-time types, which is the default for most PostgreSQL configurations. When creating a connection, ODB examines the `integer_datetimes` PostgreSQL server parameter and if it is `false`, `odb::pgsql::database_exception` is thrown. You may check the value of this parameter for your server by executing the following SQL query:

```
SHOW integer_datetimes
```

19.5.5 Timezones

ODB does not currently natively support the PostgreSQL date-time types with timezone information. However, these types can be accessed by mapping them to one of the natively supported types, as discussed in Section 14.8, "Database Type Mapping Pragmas".

19.5.6 `NUMERIC` Type Support

Support for the PostgreSQL `NUMERIC` type is limited to providing a binary buffer containing the binary representation of the value. For more information on the binary format used to store `NUMERIC` values refer to the PostgreSQL documentation. An alternative approach to accessing `NUMERIC` values is to map this type to one of the natively supported ones, as discussed in Section 14.8, "Database Type Mapping Pragmas".

19.5.7 Bulk Operations Support

Support for bulk operations (Section 15.3, "Bulk Database Operations") requires PostgreSQL client library (`libpq`) version 14 or later and PostgreSQL server version 7.4 or later.

19.6 PostgreSQL Index Definitions

When the `index` pragma (Section 14.7, "Index Definition Pragmas") is used to define a PostgreSQL index, the `type` clause specifies the index type (for example, `UNIQUE`), the `method` clause specifies the index method (for example, `BTREE`, `HASH`, `GIN`, etc.), and the `options` clause specifies additional index options, such as storage parameters, table spaces, and the `WHERE` predicate. To support the definition of concurrent indexes, the `type` clause can end with

the word `CONCURRENTLY` (upper and lower cases are recognized). The column options can be used to specify collations, operator classes, and the sort order. For example:

```
#pragma db object
class object
{
  ...

  std::string name_;

  #pragma db index                            \
              type("UNIQUE CONCURRENTLY")     \
              method("HASH")                  \
              member(name_, "DESC")           \
              options("WITH(FILLFACTOR = 80)")
};
```

Index names in PostgreSQL are schema-global. To avoid name clashes, ODB automatically prefixes each index name with the table name on which it is defined.

19.7 PostgreSQL Stored Procedures and Functions

ODB native views (Section 10.6, "Native Views") can be used to call PostgreSQL stored procedures and functions. For example, assuming we are using the `person` class from Chapter 2, "Hello World Example" (and the corresponding `person` table), we can create a stored function that given the min and max ages returns some information about all the people in that range:

```
CREATE FUNCTION
person_range (IN min_age INTEGER,
              IN max_age INTEGER,
              OUT age SMALLINT,
              OUT first TEXT,
              OUT last TEXT)
RETURNS SETOF RECORD AS $$
  SELECT age, first, last
    FROM person
    WHERE age >= min_age AND age <= max_age;
$$ LANGUAGE SQL STABLE;
```

Given the above stored function we can then define an ODB view that can be used to call it and retrieve its result:

```
#pragma db view query("/*CALL*/ SELECT * FROM person_range((?))")
struct person_range
{
  unsigned short age;
  std::string first;
  std::string last;
};
```

Notice the special `/*CALL*/` prefix: because PostgreSQL uses ordinary `SELECT` queries to call functions, we need to communicate to ODB that the query is actually a function call.

The following example shows how we can use the above view to print the list of people in a specific age range:

```
using query = odb::query<person_range>;
using result = odb::result<person_range>;

transaction t (db.begin ());

result r (
  db.query<person_range> (
    query::_val (1) + "," + query::_val (18)));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cerr << i->first << " " << i->last << " " << i->age << endl;

t.commit ();
```

Note that as with all native views, the order and types of data members must match those returned by the stored function or procedure.

In the above example, the stored function returned a set of rows. Other common cases are functions that return a single row, a single value, or nothing (`VOID`). An example of a function returning a single row via the `OUT` parameters:

```
CREATE FUNCTION
person_age_range (IN last_name TEXT,
                  OUT min_age SMALLINT,
                  OUT max_age SMALLINT)
AS $$
  SELECT min(age), max(age)
    FROM person
    WHERE last = last_name;
$$ LANGUAGE SQL STABLE;
```

```
#pragma db view query("/*CALL*/ SELECT * FROM person_age_range((?))")
struct person_age_range
{
  unsigned short min_age;
  unsigned short max_age;
};

using query = odb::query<person_age_range>;

person_age_range r (
  db.query_value<person_age_range> (
    query::_val ("Doe")));
```

An example of a function returning a single value:

```
CREATE FUNCTION
person_count ()
RETURNS BIGINT AS $$
  SELECT count(id)
    FROM person;
$$ LANGUAGE SQL STABLE;

#pragma db view query("/*CALL*/ SELECT * FROM person_count()")
struct person_count
{
  unsigned long long count;
};

unsigned long long count (db.query_value<person_count> ().count);
```

Finally, a function that returns `VOID`:

```
CREATE FUNCTION
person_increment_age (IN min_age INTEGER,
                      IN max_age INTEGER)
RETURNS VOID AS $$
  UPDATE person
    SET age = age + 1
    WHERE age >= min_age AND age <= max_age;
$$ LANGUAGE SQL;

#pragma db view query("/*CALL*/ SELECT * FROM person_increment_age((?))")
struct person_increment_age {};

using query = odb::query<person_increment_age>;

db.query_value<person_increment_age> (
  query::_val (1) + ","  +
  query::_val (18));
```

We can reuse a single view to call several functions (or procedure; see below) that return `VOID` or, more generally, return the same result, by specifying the function (or procedure) at the call site rather than in the view definition. For example:

```
#pragma db view
struct no_result {};

using query = odb::query<no_result>;

db.query_one<no_result> (
  "/*CALL*/ SELECT * FROM person_increment_age(" +
  query::_val (1) + "," +
  query::_val (18) + ")");
```

In contrast to functions, stored procedures in PostgreSQL are called with the `CALL` statement. They can only return a single set of values via the `OUT` parameters. If there are no output parameters, then a stored procedure doesn't return anything. For example:

```
CREATE PROCEDURE
person_increment_age (IN id BIGINT,
                      OUT first TEXT,
                      OUT last TEXT,
                      OUT age SMALLINT)
AS $$
  UPDATE person
    SET age = age + 1
    WHERE person.id = id
    RETURNING first, last, age;
$$ LANGUAGE SQL;

#pragma db view query("CALL person_increment_age((?))")
struct person_increment_age
{
  std::string first;
  std::string last;
  unsigned short age;
};

using query = odb::query<person_increment_age>;

person_increment_age r (
  db.query_value<person_increment_age> (
    query::_val (123) + ",NULL,NULL,NULL"));
```

Note that in the `CALL` statement, output parameters must be included in the argument list, normally as `NULL`.

A stored procedure that doesn't return anything and has no parameters can be called with `execute()` (3.12 Executing Native SQL Statements) rather than `query()`. Such a procedure can also control transactions. For example:

```
CREATE PROCEDURE
person_increment_all_ages ()
AS $$
BEGIN
  UPDATE person
    SET age = age + 1;
  COMMIT;
END;
$$ LANGUAGE PLPGSQL;

db.connection ()->execute ("CALL person_increment_all_ages()");
```

For more information on how to call stored procedures and functions with the various return approaches refer to the `pgsql/stored-proc` test in the `odb-tests` package.

20 Oracle Database

To generate support code for the Oracle database you will need to pass the "`--database oracle`" (or "`-d oracle`") option to the ODB compiler. Your application will also need to link to the Oracle ODB runtime library (`libodb-oracle`). All Oracle-specific ODB classes are defined in the `odb::oracle` namespace.

20.1 Oracle Type Mapping

The following table summarizes the default mapping between basic C++ value types and Oracle database types. This mapping can be customized on the per-type and per-member basis using the ODB Pragma Language (Chapter 14, "ODB Pragma Language").

C++ Type	Oracle Type	Default `NULL` Semantics
`bool`	`NUMBER(1)`	`NOT NULL`
`char`	`CHAR(1)`	`NOT NULL`
`signed char`	`NUMBER(3)`	`NOT NULL`
`unsigned char`	`NUMBER(3)`	`NOT NULL`
`short`	`NUMBER(5)`	`NOT NULL`
`unsigned short`	`NUMBER(5)`	`NOT NULL`
`int`	`NUMBER(10)`	`NOT NULL`
`unsigned int`	`NUMBER(10)`	`NOT NULL`
`long`	`NUMBER(19)`	`NOT NULL`
`unsigned long`	`NUMBER(20)`	`NOT NULL`
`long long`	`NUMBER(19)`	`NOT NULL`
`unsigned long long`	`NUMBER(20)`	`NOT NULL`
`float`	`BINARY_FLOAT`	`NOT NULL`
`double`	`BINARY_DOUBLE`	`NOT NULL`
`std::string`	`VARCHAR2(512)`	`NULL`
`char[N]`	`VARCHAR2(N-1)`	`NULL`

It is possible to map the `char` C++ type to an integer database type (for example, `NUMBER(3)`) using the `db type` pragma (Section 14.4.3, "`type`").

In Oracle empty `VARCHAR2` and `NVARCHAR2` strings are represented as a `NULL` value. As a result, columns of the `std::string` and `char[N]` types are by default declared as `NULL` except for primary key columns. However, you can override this by explicitly declaring such columns as `NOT NULL` with the `db not_null` pragma (Section 14.4.6, "`null/not_null`"). This also means that for object ids that are mapped to these Oracle types, an empty string is an invalid value.

Additionally, by default, C++ enums and C++11 enum classes are automatically mapped to the Oracle types corresponding to their underlying integral types (see table above). The default `NULL` semantics is `NOT NULL`. For example:

```
enum color {red, green, blue};
enum class taste: unsigned char
{
  bitter = 1,
  sweet,
  sour = 4,
  salty
};

#pragma db object
class object
{
  ...

  color color_; // Automatically mapped to NUMBER(10).
  taste taste_; // Automatically mapped to NUMBER(3).
};
```

It is also possible to add support for additional Oracle types, such as `XML`, geospatial types, user-defined types, and collections (arrays, table types). For more information, refer to Section 14.8, "Database Type Mapping Pragmas".

20.1.1 String Type Mapping

The Oracle ODB runtime library provides support for mapping the `std::string`, `char[N]`, and `std::array<char, N>` types to the Oracle `CHAR`, `VARCHAR2`, `CLOB`, `NCHAR`, `NVARCHAR2`, and `NCLOB` types. However, these mappings are not enabled by default (in particular, by default, `std::array` will be treated as a container). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type ("CHAR(2)")
  char state_[2];

  #pragma db type ("VARCHAR(128)") null
  std::string name_;

  #pragma db type ("CLOB")
  std::string text_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
#pragma db value(std::string) type("VARCHAR(128)") null

#pragma db object
class object
{
  ...

  std::string name_; // Mapped to VARCHAR(128).

  #pragma db type ("CLOB")
  std::string text_; // Mapped to CLOB.
};
```

The `char[N]` and `std::array<char, N>` values may or may not be zero-terminated. When extracting such values from the database, ODB will append the zero terminator if there is enough space.

20.1.2 Binary Type Mapping

The Oracle ODB runtime library provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, `unsigned char[N]`, `std::array<char, N>`, and `std::array<unsigned char, N>` types to the Oracle `BLOB` and `RAW` types. However, these mappings are not enabled by default (in particular, by default, `std::vector` and `std::array` will be treated as containers). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...
```

```
  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("RAW(16)")
  unsigned char uuid_[16];
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using buffer = std::vector<char>;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Note also that in native queries (Chapter 4, "Querying the Database") `char[N]` and `std::array<char, N>` parameters are by default passed as a string rather than a binary. To pass such parameters as a binary, we need to specify the database type explicitly in the `_val()`/`_ref()` calls. Note also that we don't need to do this for the integrated queries, for example:

```
char u[16] = {...};

db.query<object> ("uuid = " + query::_val<odb::oracle::id_raw> (u));
db.query<object> (query::uuid == query::_ref (u));
```

20.2 Oracle Database Class

The Oracle `database` class encapsulates the OCI environment handle as well as the database connection string and user credentials that are used to establish connections to the database. It has the following interface:

```
namespace odb
{
  namespace oracle
  {
    class database: public odb::database
    {
    public:
      database (const std::string& user,
                const std::string& password,
                const std::string& db,
                ub2 charset = 0,
                ub2 ncharset = 0,
```

```
            OCIEnv* environment = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  database (const std::string& user,
            const std::string& password,
            const std::string& service,
            const std::string& host,
            unsigned int port = 0,
            ub2 charset = 0,
            ub2 ncharset = 0,
            OCIEnv* environment = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  database (int& argc,
            char* argv[],
            bool erase = false,
            ub2 charset = 0,
            ub2 ncharset = 0,
            OCIEnv* environment = 0,
            std::[auto|unique]_ptr<connection_factory> = 0);

  static void
  print_usage (std::ostream&);

public:
  const std::string&
  user () const;

  const std::string&
  password () const;

  const std::string&
  db () const;

  const std::string&
  service () const;

  const std::string&
  host () const;

  unsigned int
  port () const;

  ub2
  charset () const;

  ub2
  ncharset () const;

  OCIEnv*
  environment ();
```

```
    public:
      connection_ptr
      connection ();
    };
  }
}
```

You will need to include the `<odb/oracle/database.hxx>` header file to make this class available in your application.

The overloaded `database` constructors allow us to specify the Oracle database parameters that should be used when connecting to the database. The `db` argument in the first constructor is a connection identifier that specifies the database to connect to. For more information on the format of the connection identifier, refer to the Oracle documentation.

The second constructor allows us to specify the individual components of a connection identifier as the `service`, `host`, and `port` arguments. If the `host` argument is empty, then localhost is used by default. Similarly, if the `port` argument is zero, then the default port is used.

The last constructor extracts the database parameters from the command line. The following options are recognized:

```
--user <login>
--password <password>
--database <connect-id>
--service <name>
--host <host>
--port <integer>
--options-file <file>
```

The `--options-file` option allows us to specify some or all of the database options in a file with each option appearing on a separate line followed by a space and an option value. Note that it is invalid to specify the `--database` option together with `--service`, `--host`, or `--port` options.

If the `erase` argument to this constructor is true, then the above options are removed from the `argv` array and the `argc` count is updated accordingly. This is primarily useful if your application accepts other options or arguments and you would like to get the Oracle options out of the `argv` array.

This constructor throws the `odb::oracle::cli_exception` exception if the Oracle option values are missing or invalid. See section Section 20.4, "Oracle Exceptions" for more information on this exception.

The static `print_usage()` function prints the list of options with short descriptions that are recognized by this constructor.

Additionally, all the constructors have the `charset`, `ncharset`, and `environment` arguments. The `charset` argument specifies the client-side database character encoding. Character data corresponding to the `CHAR`, `VARCHAR2`, and `CLOB` types will be delivered to and received from the application in this encoding. Similarly, the `ncharset` argument specifies the client-side national character encoding. Character data corresponding to the `NCHAR`, `NVARCHAR2`, and `NCLOB` types will be delivered to and received from the application in this encoding. For the complete list of available character encoding values, refer to the Oracle documentation. Commonly used encoding values are `873` (UTF-8), `31` (ISO-8859-1), and `1000` (UTF-16). If the database character encoding is not specified, then the `NLS_LANG` environment/registry variable is used. Similarly, if the national character encoding is not specified, then the `NLS_NCHAR` environment/registry variable is used. For more information on character encodings, refer to the `OCIEnvNlsCreate()` function in the Oracle Call Interface (OCI) documentation.

The `environment` argument allows us to provide a custom OCI environment handle. If this argument is not `NULL`, then the passed handle is used in all the OCI function calls made by this `database` class instance. Note also that the `database` instance does not assume ownership of the passed environment handle and this handle should be valid for the lifetime of the `database` instance. If a custom environment handle is used, then the `charset` and `ncharset` arguments have no effect.

The last argument to all of the constructors is a pointer to the connection factory. In C++98/03, it is `std::auto_ptr` while in C++11 `std::unique_ptr` is used instead. If we pass a non-`NULL` value, the database instance assumes ownership of the factory instance. The connection factory interface as well as the available implementations are described in the next section.

The set of accessor functions following the constructors allows us to query the parameters of the `database` instance.

The `connection()` function returns a pointer to the Oracle database connection encapsulated by the `odb::oracle::connection` class. For more information on `oracle::connection`, refer to Section 20.3, "Oracle Connection and Connection Factory".

20.3 Oracle Connection and Connection Factory

The `oracle::connection` class has the following interface:

```
namespace odb
{
  namespace oracle
  {
    class connection: public odb::connection
```

```
    {
    public:
      connection (database&);
      connection (database&, OCISvcCtx*);

      OCISvcCtx*
      handle ();

      OCIError*
      error_handle ();

      details::buffer&
      lob_buffer ();
    };

    using connection_ptr = details::shared_ptr<connection>;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.6, "Connections". The first overloaded `oracle::connection` constructor creates a new OCI service context. The OCI statement caching is enabled for the underlying session while the OCI connection pooling and session pooling are not used. The second constructor allows us to create a `connection` instance by providing an already connected Oracle service context. Note that the `connection` instance assumes ownership of this handle. The `handle()` accessor returns the OCI service context handle associated with the `connection` instance.

An OCI error handle is allocated for each `connection` instance and is available via the `error_handle()` accessor function.

Additionally, each `connection` instance maintains a large object (LOB) buffer. This buffer is used by the Oracle ODB runtime as an intermediate storage for piecewise handling of LOB data. By default, the LOB buffer has zero initial capacity and is expanded to 4096 bytes when the first LOB operation is performed. If your application requires a bigger or smaller LOB buffer, you can specify a custom capacity using the `lob_buffer()` accessor.

The `oracle::connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace oracle
  {
    class connection_factory
    {
    public:
      virtual void
      database (database&) = 0;
```

```
      virtual connection_ptr
      connect () = 0;
    };
  }
}
```

The `database()` function is called when a connection factory is associated with a database instance. This happens in the `odb::oracle::database` class constructors. The `connect()` function is called whenever a database connection is requested.

The two implementations of the `connection_factory` interface provided by the Oracle ODB runtime are `new_connection_factory` and `connection_pool_factory`. You will need to include the `<odb/oracle/connection-factory.hxx>` header file to make the `connection_factory` interface and these implementation classes available in your application.

The `new_connection_factory` class creates a new connection whenever one is requested. When a connection is no longer needed, it is released and closed. The `new_connection_factory` class has the following interface:

```
namespace odb
{
  namespace oracle
  {
    class new_connection_factory: public connection_factory
    {
    public:
      new_connection_factory ();
    };
};
```

The `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace oracle
  {
    class connection_pool_factory: public connection_factory
    {
    public:
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0);

    protected:
      class pooled_connection: public connection
      {
      public:
        pooled_connection (database_type&);
```

```
        pooled_connection (database_type&, OCISvcCtx*);
      };

      using pooled_connection_ptr = details::shared_ptr<pooled_connection>;

      virtual pooled_connection_ptr
      create ();
    };
};
```

The `max_connections` argument in the `connection_pool_factory` constructor specifies the maximum number of concurrent connections that this pool factory will maintain. Similarly, the `min_connections` argument specifies the minimum number of available connections that should be kept open.

Whenever a connection is requested, the pool factory first checks if there is an unused connection that can be returned. If there is none, the pool factory checks the `max_connections` value to see if a new connection can be created. If the total number of connections maintained by the pool is less than this value, then a new connection is created and returned. Otherwise, the caller is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked callers waiting for a connection. If so, then one of them is unblocked and is given the connection. Otherwise, the pool factory checks whether the total number of connections maintained by the pool is greater than the `min_connections` value. If that's the case, the connection is closed. Otherwise, the connection is added to the pool of available connections to be returned on the next request. In other words, if the number of connections maintained by the pool exceeds `min_connections` and there are no callers waiting for a new connection, the pool will close the excess connections.

If the `max_connections` value is 0, then the pool will create a new connection whenever all of the existing connections are in use. If the `min_connections` value is 0, then the pool will never close a connection and instead maintain all the connections that were ever created.

The `create()` virtual function is called whenever the pool needs to create a new connection. By deriving from the `connection_pool_factory` class and overriding this function we can implement custom connection establishment and configuration.

If you pass `NULL` as the connection factory to one of the `database` constructors, then the `connection_pool_factory` instance will be created by default with the min and max connections values set to `0`. The following code fragment shows how we can pass our own connection factory instance:

```
#include <odb/database.hxx>

#include <odb/oracle/database.hxx>
#include <odb/oracle/connection-factory.hxx>
```

```
int
main (int argc, char* argv[])
{
  unique_ptr<odb::oracle::connection_factory> f (
    new odb::oracle::connection_pool_factory (20));

  unique_ptr<odb::database> db (
    new oracle::database (argc, argv, false, 0, 0, 0, f));
}
```

20.4 Oracle Exceptions

The Oracle ODB runtime library defines the following Oracle-specific exceptions:

```
namespace odb
{
  namespace oracle
  {
    class database_exception: odb::database_exception
    {
    public:
      class record
      {
      public:
        sb4
        error () const;

        const std::string&
        message () const;
      };

      using records = std::vector<record>;

      using size_type = records::size_type;
      using iterator = records::const_iterator;

      iterator
      begin () const;

      iterator
      end () const;

      size_type
      size () const;

      virtual const char*
      what () const throw ();
    };
```

```
    class cli_exception: odb::exception
    {
    public:
      virtual const char*
      what () const throw ();
    };

    class invalid_oci_handle: odb::exception
    {
    public:
      virtual const char*
      what () const throw ();
    };
  }
}
```

You will need to include the `<odb/oracle/exceptions.hxx>` header file to make these exceptions available in your application.

The `odb::oracle::database_exception` is thrown if an Oracle database operation fails. The Oracle-specific error information is stored as a series of records, each containing the error code as a signed 4-byte integer and the message string. All this information is also combined and returned in a human-readable form by the `what()` function.

The `odb::oracle::cli_exception` is thrown by the command line parsing constructor of the `odb::oracle::database` class if the Oracle option values are missing or invalid. The `what()` function returns a human-readable description of an error.

The `odb::oracle::invalid_oci_handle` is thrown if an invalid handle is passed to an OCI function or if an OCI function was unable to allocate a handle. The former normally indicates a programming error while the latter indicates an out of memory condition. The `what()` function returns a human-readable description of an error.

20.5 Oracle Limitations

The following sections describe Oracle-specific limitations imposed by the current Oracle and ODB runtime versions.

20.5.1 Identifier Truncation

Oracle limits the length of database identifiers (table, column, etc., names) to 30 characters. The ODB compiler automatically truncates any identifier that is longer than 30 characters. This, however, can lead to duplicate names. A common symptom of this problem are errors during the database schema creation indicating that a database object with the same name already exists. To resolve this problem we can assign custom, shorter identifiers using the `db table` and `db column` pragmas (Chapter 14, "ODB Pragma Language"). For example:

```
#pragma db object
class long_class_name
{
  ...

  std::vector<int> long_container_x_;
  std::vector<int> long_container_y_;
};
```

In the above example, the names of the two container tables will be `long_class_name_long_container_x_` and `long_class_name_long_container_y_`. However, when truncated to 30 characters, they both become `long_class_name_long_container`. To resolve this collision we can assign a custom table name for each container:

```
#pragma db object
class long_class_name
{
  ...

  #pragma db table("long_class_name_cont_x")
  std::vector<int> long_container_x_;

  #pragma db table("long_class_name_cont_y")
  std::vector<int> long_container_y_;
};
```

20.5.2 Query Result Caching

Oracle ODB runtime implementation does not perform query result caching (Section 4.4, "Query Result") even when explicitly requested. The OCI API supports interleaving execution of multiple prepared statements on a single connection. As a result, with OCI, it is possible to have multiple uncached results and calls to other database functions do not invalidate them. The only limitation of the uncached Oracle results is the unavailability of the `result::size()` function. If you call this function on an Oracle query result, then the `odb::result_not_cached` exception (Section 3.14, "ODB Exceptions") is always thrown. Future versions of the Oracle ODB runtime library may add support for result caching.

20.5.3 Foreign Key Constraints

ODB assumes the standard SQL behavior which requires that foreign key constraints checking is deferred until the transaction is committed. Default Oracle behavior is to check such constraints immediately. As a result, when used with ODB, a custom database schema that defines foreign key constraints may need to declare such constraints as `INITIALLY DEFERRED`, as shown in the following example. By default, schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer NUMBER(20) REFERENCES Employer(id)
           DEFERRABLE INITIALLY DEFERRED);
```

You can override the default behavior and instruct the ODB compiler to generate non-deferrable foreign keys by specifying the `--fkeys-deferrable-mode not_deferrable` ODB compiler option. Note, however, that in this case the order in which you persist, update, and erase objects within a transaction becomes important.

20.5.4 Unique Constraint Violations

Due to the granularity of the Oracle error codes, it is impossible to distinguish between the duplicate primary key and other unique constraint violations. As a result, when making an object persistent, the Oracle ODB runtime will translate all unique constraint violation errors to the `object_already_persistent` exception (Section 3.14, "ODB Exceptions").

20.5.5 Large `FLOAT` and `NUMBER` Types

The Oracle `FLOAT` type with a binary precision greater than 53 and fixed-point `NUMBER` type with a decimal precision greater than 15 cannot be automatically extracted into the C++ `float` and `double` types. Instead, the Oracle ODB runtime uses a 21-byte buffer containing the binary representation of a value as an image type for such `FLOAT` and `NUMBER` types. In order to convert them into an application-specific large number representation, you will need to provide a suitable `value_traits` template specialization. For more information on the binary format used to store the `FLOAT` and `NUMBER` values, refer to the Oracle Call Interface (OCI) documentation.

An alternative approach to accessing large `FLOAT` and `NUMBER` values is to map these type to one of the natively supported ones, as discussed in Section 14.8, "Database Type Mapping Pragmas".

Note that a `NUMBER` type that is used to represent a floating point number (declared by specifying `NUMBER` without any range and scale) can be extracted into the C++ `float` and `double` types.

20.5.6 Timezones

ODB does not currently support the Oracle date-time types with timezone information. However, these types can be accessed by mapping them to one of the natively supported types, as discussed in Section 14.8, "Database Type Mapping Pragmas".

20.5.7 LONG Types

ODB does not support the deprecated Oracle `LONG` and `LONG RAW` data types. However, these types can be accessed by mapping them to one of the natively supported types, as discussed in Section 14.8, "Database Type Mapping Pragmas".

20.5.8 LOB Types and By-Value Accessors/Modifiers

As discussed in Section 14.4.5, "`get/set/access`", by-value accessor and modifier expressions cannot be used with data members of Oracle large object (LOB) data types: `BLOB`, `CLOB`, and `NCLOB`. The Oracle ODB runtime uses streaming for reading/writing LOB data directly from/to data members. As a result, by-reference accessors and modifiers should be used for these data types.

20.5.9 Database Schema Evolution

In Oracle, the type of the `name` column in the `schema_version` table is `VARCHAR2(512)`. Because this column is a primary key and `VARCHAR2` represents empty strings as `NULL` values, it is impossible to store an empty string in this column, which is what is used to represent the default schema name. As a result, in Oracle, the empty schema name is stored as a string containing a single space character. ODB performs all the necessary translations automatically and normally you do not need to worry about this implementation detail unless you are querying or modifying the `schema_version` table directly.

20.6 Oracle Index Definitions

When the `index` pragma (Section 14.7, "Index Definition Pragmas") is used to define an Oracle index, the `type` clause specifies the index type (for example, `UNIQUE`, `BITMAP`), the `method` clause is not used, and the `options` clause specifies additional index properties, such as partitioning, table spaces, etc. The column options can be used to specify the sort order. For example:

```
#pragma db object
class object
{
  ...

  std::string name_;

  #pragma db index                      \
             type("BITMAP")             \
             member(name_, "DESC")      \
             options("TABLESPACE TBS1")
};
```

Index names in Oracle are schema-global. To avoid name clashes, ODB automatically prefixes each index name with the table name on which it is defined.

21 Microsoft SQL Server Database

To generate support code for the SQL Server database you will need to pass the "`--database mssql`" (or "`-d mssql`") option to the ODB compiler. Your application will also need to link to the SQL Server ODB runtime library (`libodb-mssql`). All SQL Server-specific ODB classes are defined in the `odb::mssql` namespace.

21.1 SQL Server Type Mapping

The following table summarizes the default mapping between basic C++ value types and SQL Server database types. This mapping can be customized on the per-type and per-member basis using the ODB Pragma Language (Chapter 14, "ODB Pragma Language").

C++ Type	SQL Server Type	Default NULL Semantics
bool	BIT	NOT NULL
char	CHAR(1)	NOT NULL
signed char	TINYINT	NOT NULL
unsigned char	TINYINT	NOT NULL
short	SMALLINT	NOT NULL
unsigned short	SMALLINT	NOT NULL
int	INT	NOT NULL
unsigned int	INT	NOT NULL
long	BIGINT	NOT NULL
unsigned long	BIGINT	NOT NULL
long long	BIGINT	NOT NULL
unsigned long long	BIGINT	NOT NULL
float	REAL	NOT NULL
double	FLOAT	NOT NULL
std::string	VARCHAR(512)/VARCHAR(256)	NOT NULL
char[N]	VARCHAR(N-1)	NOT NULL
std::wstring	NVARCHAR(512)/NVARCHAR(256)	NOT NULL
wchar_t[N]	NVARCHAR(N-1)	NOT NULL
GUID	UNIQUEIDENTIFIER	NOT NULL

It is possible to map the `char` C++ type to an integer database type (for example, `TINYINT`) using the `db type` pragma (Section 14.4.3, "`type`").

Note that the `std::string` and `std::wstring` types are mapped differently depending on whether a member of one of these types is an object id or not. If the member is an object id, then for this member `std::string` is mapped to `VARCHAR(256)` and `std::wstring` — to `NVARCHAR(256)`. Otherwise, `std::string` is mapped to `VARCHAR(512)` and `std::wstring` — to `NVARCHAR(512)`. Note also that you can always change this mapping

using the `db type` pragma (Section 14.4.3, "`type`").

Additionally, by default, C++ enums and C++11 enum classes are automatically mapped to the SQL Server types corresponding to their underlying integral types (see table above). The default `NULL` semantics is `NOT NULL`. For example:

```
enum color {red, green, blue};
enum class taste: unsigned char
{
  bitter = 1,
  sweet,
  sour = 4,
  salty
};

#pragma db object
class object
{
  ...

  color color_; // Automatically mapped to INT.
  taste taste_; // Automatically mapped to TINYINT.
};
```

Note also that because SQL Server does not support unsigned integers, the `unsigned short`, `unsigned int`, and `unsigned long`/`unsigned long long` C++ types are by default mapped to the `SMALLINT`, `INT`, and `BIGINT` SQL Server types, respectively. The sign bit of the value stored by the database for these types will contain the most significant bit of the actual unsigned value being persisted. Similarly, because there is no signed version of the `TINYINT` SQL Server type, by default, the `signed char` C++ type is mapped to `TINYINT`. As a result, the most significant bit of the value stored by the database for this type will contain the sign bit of the actual signed value being persisted.

It is also possible to add support for additional SQL Server types, such as geospatial types, `XML`, and user-defined types. For more information, refer to Section 14.8, "Database Type Mapping Pragmas".

21.1.1 String Type Mapping

The SQL Server ODB runtime library provides support for mapping the `std::string`, `char[N]`, and `std::array<char, N>` types to the SQL Server `CHAR`, `VARCHAR`, and `TEXT` types as well as the `std::wstring`, `wchar_t[N]`, and `std::array<wchar_t, N>` types to `NCHAR`, `NVARCHAR`, and `NTEXT`. However, these mappings are not enabled by default (in particular, by default, `std::array` will be treated as a container). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...

  #pragma db type ("CHAR(2)")
  char state_[2];

  #pragma db type ("NVARCHAR(max)")
  std::wstring text_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
#pragma db value(std::wstring) type("NVARCHAR(max)")

#pragma db object
class object
{
  ...

  std::wstring text_; // Mapped to NVARCHAR(max).
};
```

The `char[N]`, `std::array<char, N>`, `wchar_t[N]`, and `std::array<wchar_t, N>` values may or may not be zero-terminated. When extracting such values from the database, ODB will append the zero terminator if there is enough space.

See also Section 21.1.4, "Long String and Binary Types" for certain limitations of long string types.

21.1.2 Binary Type and `UNIQUEIDENTIFIER` Mapping

The SQL Server ODB runtime library also provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, `unsigned char[N]`, `std::array<char, N>`, and `std::array<unsigned char, N>` types to the SQL Server `BINARY`, `VARBINARY`, and `IMAGE` types. There is also support for mapping the `char[16]` array to the SQL Server `UNIQUEIDENTIFIER` type. However, these mappings are not enabled by default (in particular, by default, `std::vector` and `std::array` will be treated as containers). To enable the alternative mappings for these types we need to specify the database type explicitly using the `db type` pragma (Section 14.4.3, "`type`"), for example:

```
#pragma db object
class object
{
  ...
```

```
  #pragma db type("UNIQUEIDENTIFIER")
  char uuid_[16];

  #pragma db type("VARBINARY(max)")
  std::vector<char> buf_;

  #pragma db type("BINARY(256)")
  unsigned char data_[256];
};
```

Alternatively, this can be done on the per-type basis, for example:

```
using buffer = std::vector<char>;
#pragma db value(buffer) type("VARBINARY(max)")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to VARBINARY(max).
};
```

Note also that in native queries (Chapter 4, "Querying the Database") `char[N]` and `std::array<char, N>` parameters are by default passed as a string rather than a binary. To pass such parameters as a binary, we need to specify the database type explicitly in the `_val()`/`_ref()` calls. Note also that we don't need to do this for the integrated queries, for example:

```
char u[16] = {...};

db.query<object> ("uuid = " + query::_val<odb::mssql::id_binary> (u));
db.query<object> (
  "uuid = " + query::_val<odb::mssql::id_uniqueidentifier> (u));
db.query<object> (query::uuid == query::_ref (u));
```

See also Section 21.1.4, "Long String and Binary Types" for certain limitations of long binary types.

21.1.3 `ROWVERSION` Mapping

`ROWVERSION` is a special SQL Server data type that is automatically incremented by the database server whenever a row is inserted or updated. As such, it is normally used to implement optimistic concurrency and ODB provides support for using `ROWVERSION` instead of the more portable approach for optimistic concurrency (Chapter 12, "Optimistic Concurrency").

`ROWVERSION` is a 64-bit value which is mapped by ODB to `unsigned long long`. As a result, to use `ROWVERSION` for optimistic concurrency we need to make sure that the version column is of the `unsigned long long` type. We also need to explicitly specify that it should be mapped to the `ROWVERSION` data type. For example:

```
#pragma db object optimistic
class person
{
  ...

  #pragma db version type("ROWVERSION")
  unsigned long long version_;
};
```

21.1.4 Long String and Binary Types

For SQL Server, ODB handles character, national character, and binary data in two different ways depending on its maximum length. If the maximum length (in bytes) is less than or equal to the limit specified with the `--mssql-short-limit` ODB compiler option (1024 by default), then it is treated as *short data*, otherwise — *long data*. For short data ODB pre-allocates an intermediate buffer of the maximum size and binds it directly to a parameter or result column. This way the underlying database API (ODBC) can read/write directly from/to this buffer. In the case of long data, the data is read/written in chunks using the `SQLGetData()`/`SQLPutData()` ODBC functions. While the long data approach reduces the amount of memory used by the application, it may require greater CPU resources.

Long data has a number of limitations. In particular, when setting a custom short data limit, make sure that it is sufficiently large so that no object id in the application is treated as long data. It is also impossible to load an object or view with long data more than once as part of a query result iteration (Section 4.4, "Query Result"). Any such attempt will result in the `odb::mssql::long_data_reload` exception (Section 21.4, "SQL Server Exceptions"). For example:

```
#pragma db object
class object
{
  ...

  int num_;

  #pragma db type("VARCHAR(max)") // Long data.
  std::string str_;
};

using query = odb::query<object>;
using result = odb::result<object>;
```

```
transaction t (db.begin ());

result r (db.query<object> (query::num < 100));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  if (!i->str_.empty ()) // First load.
  {
    object o;
    i.load (o); // Error: second load, long_data_reload is thrown.
  }
}

t.commit ();
```

Finally, if a native view (Section 10.6, "Native Views") contains one or more long data members, then such members should come last both in the select-list of the native SQL query and the list of data members in the C++ class.

21.2 SQL Server Database Class

The SQL Server `database` class encapsulates the ODBC environment handle as well as the server instance address and user credentials that are used to establish connections to the database. It has the following interface:

```
namespace odb
{
  namespace mssql
  {
    enum protocol
    {
      protocol_auto,
      protocol_tcp, // TCP/IP.
      protocol_lpc, // Shared memory (local procedure call).
      protocol_np   // Named pipes.
    };

    enum transaction_isolation
    {
      isolation_read_uncommitted,
      isolation_read_committed,   // SQL Server default.
      isolation_repeatable_read,
      isolation_snapshot,
      isolation_serializable
    };

    class database: public odb::database
    {
    public:
```

```
using protocol_type = protocol;
using transaction_isolation_type = transaction_isolation;

database (const std::string& user,
          const std::string& password,
          const std::string& db,
          const std::string& server,
          const std::string& driver = "",
          const std::string& extra_connect_string = "",
          transaction_isolation_type = isolation_read_committed,
          SQLHENV environment = 0,
          std::[auto|unique]_ptr<connection_factory> = 0);

database (const std::string& user,
          const std::string& password,
          const std::string& db,
          protocol_type protocol = protocol_auto,
          const std::string& host = "",
          const std::string& instance = "",
          const std::string& driver = "",
          const std::string& extra_connect_string = "",
          transaction_isolation_type = isolation_read_committed,
          SQLHENV environment = 0,
          std::[auto|unique]_ptr<connection_factory> = 0);

database (const std::string& user,
          const std::string& password,
          const std::string& db,
          const std::string& host,
          unsigned int port,
          const std::string& driver = "",
          const std::string& extra_connect_string = "",
          transaction_isolation_type = isolation_read_committed,
          SQLHENV environment = 0,
          std::[auto|unique]_ptr<connection_factory> = 0);

database (const std::string& connect_string,
          transaction_isolation_type = isolation_read_committed,
          SQLHENV environment = 0,
          std::[auto|unique]_ptr<connection_factory> = 0);

database (int& argc,
          char* argv[],
          bool erase = false,
          const std::string& extra_connect_string = "",
          transaction_isolation_type = isolation_read_committed,
          SQLHENV environment = 0,
          std::[auto|unique]_ptr<connection_factory> = 0);

static void
print_usage (std::ostream&);
```

```
    public:
      const std::string&
      user () const;

      const std::string&
      password () const;

      const std::string&
      db () const;

      protocol_type
      protocol () const;

      const std::string&
      host () const;

      const std::string&
      instance () const;

      unsigned int
      port () const;

      const std::string&
      server () const;

      const std::string&
      driver () const;

      const std::string&
      extra_connect_string () const;

      transaction_isolation_type
      transaction_isolation () const;

      const std::string&
      connect_string () const;

      SQLHENV
      environment ();

    public:
      connection_ptr
      connection ();
    };
  }
}
```

You will need to include the `<odb/mssql/database.hxx>` header file to make this class available in your application.

The overloaded `database` constructors allow us to specify the SQL Server database parameters that should be used when connecting to the database. The `user` and `password` arguments specify the login name and password. If `user` is empty, then Windows authentication is used and the `password` argument is ignored. The `db` argument specifies the database name to open. If it is empty, then the default database for the user is used.

The `server` argument in the first constructor specifies the SQL Server instance address in the standard SQL Server address format:

`[`*`protocol`*`:]`*`host`*`[\`*`instance`*`][,`*`port`*`]`

Where *`protocol`* can be `tcp` (TCP/IP), `lpc` (shared memory), or `np` (named pipe). If protocol is not specified, then a suitable protocol is automatically selected based on the SQL Server Native Client configuration. The *`host`* component can be a host name or an IP address. If *`instance`* is not specified, then the default SQL Server instance is assumed. If port is not specified, then the default SQL Server port is used (1433). Note that you would normally specify either the instance name or the port, but not both. If both are specified, then the instance name is ignored by the SQL Server Native Client ODBC driver. For more information on the format of the SQL Server address, refer to the SQL Server Native Client ODBC driver documentation.

The second and third constructors allow us to specify all these address components (protocol, host, instance, and port) as separate arguments. The third constructor always connects using TCP/IP to the specified host and port.

The `driver` argument specifies the SQL Server Native Client ODBC driver that should be used to connect to the database. If not specified, then the latest available version is used. The following examples show common ways of connecting to the database using the first three constructors:

```
// Connect to the default SQL Server instance on the local machine
// using the default protocol. Login as 'test' with password 'secret'
// and open the 'example_db' database.
//
odb::mssql::database db1 ("test",
                          "secret",
                          "example_db");

// As above except use Windows authentication and open the default
// database for this user.
//
odb::mssql::database db2 ("",
                          "",
                          "");
```

```
// Connect to the default SQL Server instance on 'onega' using the
// default protocol. Login as 'test' with password 'secret' and open
// the 'example_db' database.
//
odb::mssql::database db3 ("test",
                          "secret",
                          "example_db"
                          "onega");

// As above but connect to the 'production' SQL Server instance.
//
odb::mssql::database db4 ("test",
                          "secret",
                          "example_db"
                          "onega\\production");

// Same as above but specify protocol, host, and instance as separate
// arguments.
//
odb::mssql::database db5 ("test",
                          "secret",
                          "example_db",
                          odb::mssql::protocol_auto,
                          "onega",
                          "production");

// As above, but use TCP/IP as the protocol.
//
odb::mssql::database db6 ("test",
                          "secret",
                          "example_db"
                          "tcp:onega\\production");

// Same as above but using separate arguments.
//
odb::mssql::database db7 ("test",
                          "secret",
                          "example_db",
                          odb::mssql::protocol_tcp,
                          "onega",
                          "production");

// As above, but use TCP/IP port instead of the instance name.
//
odb::mssql::database db8 ("test",
                          "secret",
                          "example_db"
                          "tcp:onega,1435");

// Same as above but using separate arguments. Note that here we
// don't need to specify protocol explicitly since it can only
```

```
// be TCP/IP.
//
odb::mssql::database db9 ("test",
                          "secret",
                          "example_db",
                          "onega",
                          1435);

// As above but use the specific SQL Server Native Client ODBC
// driver version.
//
odb::mssql::database dbA ("test",
                          "secret",
                          "example_db"
                          "tcp:onega,1435",
                          "SQL Server Native Client 10.0");
```

The fourth constructor allows us to pass a custom ODBC connection string that provides all the information necessary to connect to the database. Note also that all the other constructors have the `extra_connect_string` argument which can be used to specify additional ODBC connection attributes. For more information on the format of the ODBC connection string, refer to the SQL Server Native Client ODBC driver documentation.

The last constructor extracts the database parameters from the command line. The following options are recognized:

```
--user | -U <login>
--password | -P <password>
--database | -d <name>
--server | -S <address>
--driver <name>
--options-file <file>
```

The `--options-file` option allows us to specify some or all of the database options in a file with each option appearing on a separate line followed by a space and an option value.

If the `erase` argument to this constructor is true, then the above options are removed from the `argv` array and the `argc` count is updated accordingly. This is primarily useful if your application accepts other options or arguments and you would like to get the SQL Server options out of the `argv` array.

This constructor throws the `odb::mssql::cli_exception` exception if the SQL Server option values are missing or invalid. See section Section 21.4, "SQL Server Exceptions" for more information on this exception.

The static `print_usage()` function prints the list of options with short descriptions that are recognized by this constructor.

Additionally, all the constructors have the `transaction_isolation` and `environment` arguments. The `transaction_isolation` argument allows us to specify an alternative transaction isolation level that should be used by all the connections created by this database instance. The `environment` argument allows us to provide a custom ODBC environment handle. If this argument is not `NULL`, then the passed handle is used in all the ODBC function calls made by this `database` instance. Note also that the `database` instance does not assume ownership of the passed environment handle and this handle should be valid for the lifetime of the `database` instance.

The last argument to all of the constructors is a pointer to the connection factory. In C++98/03, it is `std::auto_ptr` while in C++11 `std::unique_ptr` is used instead. If we pass a non-`NULL` value, the database instance assumes ownership of the factory instance. The connection factory interface as well as the available implementations are described in the next section.

The set of accessor functions following the constructors allows us to query the parameters of the `database` instance.

The `connection()` function returns a pointer to the SQL Server database connection encapsulated by the `odb::mssql::connection` class. For more information on `mssql::connection`, refer to Section 21.3, "SQL Server Connection and Connection Factory".

21.3 SQL Server Connection and Connection Factory

The `mssql::connection` class has the following interface:

```
namespace odb
{
  namespace mssql
  {
    class connection: public odb::connection
    {
    public:
      connection (database&);
      connection (database&, SQLHDBC handle);

      SQLHDBC
      handle ();

      details::buffer&
      long_data_buffer ();
    };
```

```
    using connection_ptr = details::shared_ptr<connection>;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.6, "Connections". The first overloaded `mssql::connection` constructor creates a new ODBC connection. The created connection is configured to use the manual commit mode with multiple active result sets (MARS) enabled. The second constructor allows us to create a `connection` instance by providing an already established ODBC connection. Note that the `connection` instance assumes ownership of this handle. The `handle()` accessor returns the underlying ODBC connection handle associated with the `connection` instance.

Additionally, each `connection` instance maintains a long data buffer. This buffer is used by the SQL Server ODB runtime as an intermediate storage for piecewise handling of long data. By default, the long data buffer has zero initial capacity and is expanded to 4096 bytes when the first long data operation is performed. If your application requires a bigger or smaller long data buffer, you can specify a custom capacity using the `long_data_buffer()` accessor.

The `mssql::connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace mssql
  {
    class connection_factory
    {
    public:
      virtual void
      database (database&) = 0;

      virtual connection_ptr
      connect () = 0;
    };
  }
}
```

The `database()` function is called when a connection factory is associated with a database instance. This happens in the `odb::mssql::database` class constructors. The `connect()` function is called whenever a database connection is requested.

The two implementations of the `connection_factory` interface provided by the SQL Server ODB runtime are `new_connection_factory` and `connection_pool_factory`. You will need to include the `<odb/mssql/connection-factory.hxx>` header file to make the `connection_factory` interface and these implementation classes available in your application.

The `new_connection_factory` class creates a new connection whenever one is requested. When a connection is no longer needed, it is released and closed. The `new_connection_factory` class has the following interface:

```
namespace odb
{
  namespace mssql
  {
    class new_connection_factory: public connection_factory
    {
    public:
      new_connection_factory ();
    };
};
```

The `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace mssql
  {
    class connection_pool_factory: public connection_factory
    {
    public:
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0);

    protected:
      class pooled_connection: public connection
      {
      public:
        pooled_connection (database_type&);
        pooled_connection (database_type&, SQLHDBC handle);
      };

      using pooled_connection_ptr = details::shared_ptr<pooled_connection>;

      virtual pooled_connection_ptr
      create ();
    };
};
```

The `max_connections` argument in the `connection_pool_factory` constructor specifies the maximum number of concurrent connections that this pool factory will maintain. Similarly, the `min_connections` argument specifies the minimum number of available connections that should be kept open.

Whenever a connection is requested, the pool factory first checks if there is an unused connection that can be returned. If there is none, the pool factory checks the `max_connections` value to see if a new connection can be created. If the total number of connections maintained by the pool is less than this value, then a new connection is created and returned. Otherwise, the caller is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked callers waiting for a connection. If so, then one of them is unblocked and is given the connection. Otherwise, the pool factory checks whether the total number of connections maintained by the pool is greater than the `min_connections` value. If that's the case, the connection is closed. Otherwise, the connection is added to the pool of available connections to be returned on the next request. In other words, if the number of connections maintained by the pool exceeds `min_connections` and there are no callers waiting for a new connection, the pool will close the excess connections.

If the `max_connections` value is 0, then the pool will create a new connection whenever all of the existing connections are in use. If the `min_connections` value is 0, then the pool will never close a connection and instead maintain all the connections that were ever created.

The `create()` virtual function is called whenever the pool needs to create a new connection. By deriving from the `connection_pool_factory` class and overriding this function we can implement custom connection establishment and configuration.

If you pass `NULL` as the connection factory to one of the `database` constructors, then the `connection_pool_factory` instance will be created by default with the min and max connections values set to `0`. The following code fragment shows how we can pass our own connection factory instance:

```
#include <odb/database.hxx>

#include <odb/mssql/database.hxx>
#include <odb/mssql/connection-factory.hxx>

int
main (int argc, char* argv[])
{
  unique_ptr<odb::mssql::connection_factory> f (
    new odb::mssql::connection_pool_factory (20));

  unique_ptr<odb::database> db (
    new mssql::database (argc, argv, false, "", 0, f));
}
```

21.4 SQL Server Exceptions

The SQL Server ODB runtime library defines the following SQL Server-specific exceptions:

```
namespace odb
{
  namespace mssql
  {
    class database_exception: odb::database_exception
    {
    public:
      class record
      {
      public:
        SQLINTEGER
        error () const;

        const std::string&
        sqlstate () const;

        const std::string&
        message () const;
      };

      using records = std::vector<record>;

      using size_type = records::size_type;
      using iterator = records::const_iterator;

      iterator
      begin () const;

      iterator
      end () const;

      size_type
      size () const;

      virtual const char*
      what () const throw ();
    };

    class cli_exception: odb::exception
    {
    public:
      virtual const char*
      what () const throw ();
    };

    class long_data_reload: odb::exception
```

```
    {
    public:
      virtual const char*
      what () const throw ();
    };
  }
}
```

You will need to include the `<odb/mssql/exceptions.hxx>` header file to make these exceptions available in your application.

The `odb::mssql::database_exception` is thrown if an SQL Server database operation fails. The SQL Server-specific error information is stored as a series of records, each containing the error code as a signed 4-byte integer, the SQLSTATE code, and the message string. All this information is also combined and returned in a human-readable form by the `what()` function.

The `odb::mssql::cli_exception` is thrown by the command line parsing constructor of the `odb::mssql::database` class if the SQL Server option values are missing or invalid. The `what()` function returns a human-readable description of an error.

The `odb::mssql::long_data_reload` is thrown if an attempt is made to re-load an object or view with long data as part of a query result iteration. For more information, refer to Section 21.1, "SQL Server Type Mapping".

21.5 SQL Server Limitations

The following sections describe SQL Server-specific limitations imposed by the current SQL Server and ODB runtime versions.

21.5.1 Query Result Caching

SQL Server ODB runtime implementation does not perform query result caching (Section 4.4, "Query Result") even when explicitly requested. The ODBC API and the SQL Server Native Client ODBC driver support interleaving execution of multiple prepared statements on a single connection. As a result, it is possible to have multiple uncached results and calls to other database functions do not invalidate them. The only limitation of the uncached SQL Server results is the unavailability of the `result::size()` function. If you call this function on an SQL Server query result, then the `odb::result_not_cached` exception (Section 3.14, "ODB Exceptions") is always thrown. Future versions of the SQL Server ODB runtime library may add support for result caching.

21.5.2 Foreign Key Constraints

ODB assumes the standard SQL behavior which requires that foreign key constraints checking is deferred until the transaction is committed. The only behavior supported by SQL Server is to check such constraints immediately. As a result, by default, schemas generated by the ODB compiler for SQL Server have foreign key definitions commented out. They are retained only for documentation.

You can override the default behavior and instruct the ODB compiler to generate non-deferrable foreign keys by specifying the `--fkeys-deferrable-mode not_deferrable` ODB compiler option. Note, however, that in this case the order in which you persist, update, and erase objects within a transaction becomes important.

21.5.3 Unique Constraint Violations

Due to the granularity of the ODBC error codes, it is impossible to distinguish between the duplicate primary key and other unique constraint violations. As a result, when making an object persistent, the SQL Server ODB runtime will translate all unique constraint violation errors to the `object_already_persistent` exception (Section 3.14, "ODB Exceptions").

21.5.4 Multi-threaded Windows Applications

Multi-threaded Windows applications must use the `_beginthread()`/`_beginthreadex()` and `_endthread()`/`_endthreadex()` CRT functions instead of the `CreateThread()` and `EndThread()` Win32 functions to start and terminate threads. This is a limitation of the ODBC implementation on Windows.

21.5.5 Affected Row Count and DDL Statements

SQL Server always returns zero as the number of affected rows for DDL statements. In particular, this means that the `database::execute()` (Section 3.12, "Executing Native SQL Statements") function will always return zero for such statements.

21.5.6 Long Data and Auto Object Ids, `ROWVERSION`

SQL Server 2005 has a bug that causes it to fail on an `INSERT` or `UPDATE` statement with the `OUTPUT` clause (used to return automatically assigned object ids as well as `ROWVERSION` values) if one of the inserted columns is long data. The symptom of this bug in ODB is an exception thrown by the `database::persist()` or `database::update()` function when used on an object that contains long data and has an automatically assigned object id or uses `ROWVERSION`-based optimistic concurrency (Section 21.1.1, "`ROWVERSION` Support"). The error message reads "This operation conflicts with another pending operation on this transaction. The operation failed."

For automatically assigned object ids ODB includes a workaround for this bug which uses a less efficient method to obtain id values for objects that contain long data. To enable this workaround you need to specify that the generated code will be used with SQL Server 2005 or later by passing the `--mssql-server-version 9.0` ODB compiler option.

For `ROWVERSION`-based optimistic concurrency no workaround is currently provided. The ODB compiler will issue an error for objects that use `ROWVERSION` for optimistic concurrency and containing long data.

21.5.7 Long Data and By-Value Accessors/Modifiers

As discussed in Section 14.4.5, "`get/set/access`", by-value accessor and modifier expressions cannot be used with data members of long data types. The SQL Server ODB runtime uses streaming for reading/writing long data directly from/to data members. As a result, by-reference accessors and modifiers should be used for these data types.

21.5.8 Bulk Update and `ROWVERSION`

The bulk update operation (Section 15.3, "Bulk Database Operations") is not yet supported for persistent classes that use `ROWVERSION`-based optimistic concurrency. For such classes the bulk `update()` function is not available. The bulk persist and erase support is still provided.

21.6 SQL Server Index Definitions

When the `index` pragma (Section 14.7, "Index Definition Pragmas") is used to define an SQL Server index, the `type` clause specifies the index type (for example, `UNIQUE`, `CLUSTERED`), the `method` clause is not used, and the `options` clause specifies additional index properties. The column options can be used to specify the sort order. For example:

```
#pragma db object
class object
{
  ...

  std::string name_;

  #pragma db index                                \
              type("UNIQUE CLUSTERED")            \
              member(name_, "DESC")               \
              options("WITH(FILLFACTOR = 80)")
};
```

21.7 SQL Server Stored Procedures

ODB native views (Section 10.6, "Native Views") can be used to call SQL Server stored procedures. For example, assuming we are using the `person` class from Chapter 2, "Hello World Example" (and the corresponding `person` table), we can create a stored procedure that given the min and max ages returns some information about all the people in that range:

```
CREATE PROCEDURE dbo.person_range (
  @min_age SMALLINT,
  @max_age SMALLINT)
AS
  SELECT age, first, last FROM person
    WHERE age >= @min_age AND age <= @max_age;
```

Given the above stored procedure we can then define an ODB view that can be used to call it and retrieve its result:

```
#pragma db view query("EXEC person_range (?)")
struct person_range
{
  unsigned short age;
  std::string first;
  std::string last;
};
```

The following example shows how we can use the above view to print the list of people in a specific age range:

```
using query = odb::query<person_range>;
using result = odb::result<person_range>;

transaction t (db.begin ());

result r (
  db.query<person_range> (
    query::_val (1) + "," + query::_val (18)));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
  cerr << i->first << " " << i->last << " " << i->age << endl;

t.commit ();
```

Note that as with all native views, the order and types of data members must match those of columns in the `SELECT` list inside the stored procedure.

There are also a number of limitations when it comes to calling SQL Server stored procedures with ODB views. There is currently no support for output parameters, however, this is planned for a future version. In the meantime, to call a stored procedure that has output parameters we

have to use a wrapper procedure that converts such parameters to a SELECT result. For example, given the following procedure that calculates the age range of the people in our database:

```
CREATE PROCEDURE dbo.person_age_range (
  @min_age SMALLINT = NULL OUTPUT,
  @max_age SMALLINT = NULL OUTPUT)
AS
  SELECT @min_age = MIN(age), @max_age = MAX(max) FROM person;
```

We can create a wrapper procedure like this:

```
CREATE PROCEDURE dbo.person_age_range_odb
AS
  DECLARE @min_age SMALLINT, @max_age SMALLINT;
  EXEC person_age_range @min_age OUTPUT, @max_age OUTPUT;
  SELECT @min_age, @max_age;
```

And a view like this:

```
#pragma db view query("EXEC person_age_range_odb")
struct person_age_range
{
  unsigned short min_age;
  unsigned short max_age;
};
```

Which we can then use to call the stored procedure:

```
transaction t (db.begin ());

person_age_range ar (db.query_value<person_age_range> ());
cerr << ar.min_age << " " << ar.max_age << endl;

t.commit ();
```

In SQL Server, a stored procedure can produce multiple results. For example, if a stored procedure executes several SELECT statements, then the result of calling such a procedure consists of multiple row sets, one for each SELECT statement. Because such multiple row sets can contain varying number and type of columns, they cannot be all extracted into a single view. Consequently, these kind of stored procedures are currently not supported.

A stored procedure may also produce no row sets at all. For example, a stored procedure that only executes DML statements would exhibit this behavior. To call such a procedure we use an empty view, for example:

```
CREATE PROCEDURE dbo.insert_person (
  @first VARCHAR(512),
  @last VARCHAR(512),
  @age SMALLINT)
AS
  INSERT INTO person(first, last, age)
    VALUES(@first, @last, @age);

#pragma db view
struct no_result {};

transaction t (db.begin ());

db.query_one<no_result> (
  "EXEC insert_person" +
    query::_val ("John") + "," +
    query::_val ("Doe") + "," +
    query::_val (21));

t.commit ();
```

Finally, an SQL Server stored procedure can also return an integer status code. Similar to output parameters, this code can only be observed by an ODB view if it is converted to a `SELECT` result. For more information on how to do this and for other examples of stored procedure calls, refer to the `mssql/stored-proc` test in the `odb-tests` package.

PART III PROFILES

Part III covers the integration of ODB with popular C++ frameworks and libraries. It consists of the following chapters.

22 Profiles Introduction

23 Boost Profile

24 Qt Profile

22 Profiles Introduction

ODB profiles are a generic mechanism for integrating ODB with widely-used C++ frameworks and libraries. A profile provides glue code which allows you to seamlessly persist various components, such as smart pointers, containers, and value types found in these frameworks or libraries. The code necessary to implement a profile is packaged into the so called profile library. For example, the Boost profile implementation is provided by the `libodb-boost` profile library.

Besides linking the profile library to our application, it is also necessary to let the ODB compiler know which profiles we are using. This is accomplished with the `--profile` (or `-p` alias) option. For example:

```
odb --profile boost ...
```

Some profiles, especially those covering frameworks or libraries that consist of multiple sub-libraries, provide sub-profiles that allow you to pick and choose which components you would like to use in your application. For example, the `boost` profile contains the `boost/data-time` sub-profile. If we are only interested in the `date_time` types, then we can pass `boost/data-time` instead of `boost` to the `--profile` option, for example:

```
odb --profile boost/date-time ...
```

To summarize, you will need to perform the following steps in order to make use of a profile in your application:

1. ODB compiler: if necessary, specify the path to the profile library headers (`-I` option).
2. ODB compiler: specify the profile you would like to use with the `--profile` option.
3. C++ compiler: if necessary, specify the path to the profile library headers (normally `-I` option).
4. Linker: link the profile library to the application.

The remaining chapters in this part of the manual describe the standard profiles provided by ODB.

23 Boost Profile

The ODB profile implementation for Boost is provided by the `libodb-boost` library and consists of multiple sub-profiles corresponding to the individual Boost libraries. To enable all the available Boost sub-profiles, pass `boost` as the profile name to the `--profile` ODB compiler option. Alternatively, you can enable only specific sub-profiles by passing individual sub-profile names to `--profile`. The following sections in this chapter discuss each Boost sub-profile in detail. The `boost` example in the `odb-examples` package shows how to enable and use the Boost profile.

Some sub-profiles may throw exceptions to indicate error conditions, such as the inability to store a specific value in a particular database system. All such exceptions derive from the `odb::boost::exception` class which in turn derives from the root of the ODB exception hierarchy, class `odb::exception` (Section 3.14, "ODB Exceptions"). The `odb::boost::exception` class is defined in the `<odb/boost/exception.hxx>` header file and has the same interface as `odb::exception`. Concrete exceptions that can be thrown by the Boost sub-profiles are described in the following sections.

23.1 Smart Pointers Library

The `smart-ptr` sub-profile provides persistence support for a subset of smart pointers from the Boost `smart_ptr` library. To enable only this profile, pass `boost/smart-ptr` to the `--profile` ODB compiler option.

The currently supported smart pointers are `boost::shared_ptr` and `boost::weak_ptr`. For more information on using smart pointers as pointers to objects and views, refer to Section 3.3, "Object and View Pointers" and Chapter 6, "Relationships". For more information on using smart pointers as pointers to values, refer to Section 7.3, "Pointers and `NULL` Value Semantics". When used as a pointer to a value, only `boost::shared_ptr` is supported. For example:

```
#pragma db object
class person
{
  ...

  #pragma db null
  boost::shared_ptr<std::string> middle_name_;
};
```

To provide finer grained control over object relationship loading, the `smart-ptr` sub-profile also provides the lazy counterparts for the above pointers:
`odb::boost::lazy_shared_ptr` and `odb::boost::lazy_weak_ptr`. You will need to include the `<odb/boost/lazy-ptr.hxx>` header file to make the lazy variants available in your application. For a description of the lazy pointer interface and semantics refer to

Section 6.4, "Lazy Pointers". The following example shows how we can use these smart pointers to establish a relationship between persistent objects.

```
class employee;

#pragma db object
class position
{
  ...

  #pragma db inverse(position_)
  odb::boost::lazy_weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

  #pragma db not_null
  boost::shared_ptr<position> position_;
};
```

Besides providing persistence support for the above smart pointers, the `smart-ptr` sub-profile also changes the default pointer (Section 3.3, "Object and View Pointers") to `boost::shared_ptr`. In particular, this means that database functions that return dynamically allocated objects and views will return them as `boost::shared_ptr` pointers. To override this behavior, add the `--default-pointer` option specifying the alternative pointer type after the `--profile` option.

23.2 Unordered Containers Library

The `unordered` sub-profile provides persistence support for the containers from the Boost `unordered` library. To enable only this profile, pass `boost/unordered` to the `--profile` ODB compiler option.

The supported containers are `boost::unordered_set`, `boost::unordered_map`, `boost::unordered_multiset`, and `boost::unordered_multimap`. For more information on using the set and multiset containers with ODB, refer to Section 5.2, "Set and Multiset Containers". For more information on using the map and multimap containers with ODB, refer to Section 5.3, "Map and Multimap Containers". The following example shows how the `unordered_set` container may be used within a persistent object.

```
#pragma db object
class person
{
  ...
  boost::unordered_set<std::string> emails_;
};
```

23.3 Multi-Index Container Library

The `multi-index` sub-profile provides persistence support for `boost::multi_index_container` from the Boost Multi-Index library. To enable only this profile, pass `boost/multi-index` to the `--profile` ODB compiler option. The following example shows how `multi_index_container` may be used within a persistent object.

```
namespace mi = boost::multi_index;

#pragma db object
class person
{
  ...

  using emails =
    mi::multi_index_container<
      std::string,
      mi::indexed_by<
        mi::sequenced<>,
        mi::ordered_unique<mi::identity<std::string>>
      >
    >;

  emails emails_;
};
```

Note that a `multi_index_container` instantiation is stored differently in the database depending on whether it has any `sequenced` or `random_access` indexes. If it does, then it is treated as an ordered container (Section 5.1, "Ordered Containers") with the first such index establishing the order. Otherwise, it is treated as a set container (Section 5.2, "Set and Multiset Containers").

Note also that there is a terminology clash between ODB and Boost Multi-Index. The ODB term *ordered container* translates to Multi-Index terms *sequenced index* and *random access index* while the ODB term *set container* translates to Multi-Index terms *ordered index* and *hashed index*.

The `emails` container from the above example is stored as an ordered container. In contrast, the following `aliases` container is stored as a set.

```
namespace mi = boost::multi_index;

#pragma db value
struct name
{
  std::string first;
  std::string last;
};

bool operator< (const name&, const name&);

#pragma db object
class person
{
  ...

  using aliases =
    mi::multi_index_container<
      name,
      mi::indexed_by<
        mi::ordered_unique<mi::identity<name>>
        mi::ordered_non_unique<
          mi::member<name, std::string, &name::first>
        >,
        mi::ordered_non_unique<
          mi::member<name, std::string, &name::last>
        >
      >
    > ;

  aliases aliases_;
};
```

23.4 Optional Library

The `optional` sub-profile provides persistence support for the `boost::optional` container from the Boost `optional` library. To enable only this profile, pass `boost/optional` to the `--profile` ODB compiler option.

In a relational database `boost::optional` is mapped to a column that can have a `NULL` value. Similar to `odb::nullable` (Section 7.3, "Pointers and `NULL` Value Semantics"), it can be used to add the `NULL` semantics to existing C++ types. For example:

```
#include <boost/optional.hpp>

#pragma db object
class person
{
  ...
```

```
  std::string first_;                     // TEXT NOT NULL
  boost::optional<std::string> middle_;   // TEXT NULL
  std::string last_;                      // TEXT NOT NULL
};
```

Note also that similar to `odb::nullable`, when this profile is used, the `NULL` values are automatically enabled for data members of the `boost::optional` type.

23.5 Date Time Library

The `date-time` sub-profile provides persistence support for a subset of types from the Boost `date_time` library. It is further subdivided into two sub-profiles, `gregorian` and `posix_time`. The `gregorian` sub-profile provides support for types from the `boost::gregorian` namespace, while the `posix-time` sub-profile provides support for types from the `boost::posix_time` namespace. To enable the entire `date-time` sub-profile, pass `boost/date-time` to the `--profile` ODB compiler option. To enable only the `gregorian` sub-profile, pass `boost/date-time/gregorian`, and to enable only the `posix-time` sub-profile, pass `boost/date-time/posix-time`.

The only type that the `gregorian` sub-profile currently supports is `gregorian::date`. The types currently supported by the `posix-time` sub-profile are `posix_time::ptime` and `posix_time::time_duration`. The manner in which these types are persisted is database system dependent and is discussed in the sub-sections that follow. The example below shows how `gregorian::date` may be used within a persistent object.

```
#pragma db object
class person
{
  ...
  boost::gregorian::date date_of_birth_;
};
```

Concrete exceptions that can be thrown by the `date-time` sub-profile implementation are presented below.

```
namespace odb
{
  namespace boost
  {
    namespace date_time
    {
      struct special_value: odb::boost::exception
      {
        virtual const char*
        what () const throw ();
      };
```

```
      struct value_out_of_range: odb::boost::exception
      {
        virtual const char*
        what () const throw ();
      };
    }
  }
}
```

You will need to include the `<odb/boost/date-time/exceptions.hxx>` header file to make these exceptions available in your application.

The `special_value` exception is thrown if an attempt is made to store a Boost date-time special value that cannot be represented in the target database. The `value_out_of_range` exception is thrown if an attempt is made to store a date-time value that is out of the target database range. The specific conditions under which these exceptions are thrown are database system dependent and are discussed in more detail in the following sub-sections.

23.5.1 MySQL Database Type Mapping

The following table summarizes the default mapping between the currently supported Boost `date_time` types and the MySQL database types.

Boost `date_time` Type	MySQL Type	Default `NULL` Semantics
`gregorian::date`	`DATE`	`NULL`
`posix_time::ptime`	`DATETIME`	`NULL`
`posix_time::time_duration`	`TIME`	`NULL`

The Boost special value `date_time::not_a_date_time` is stored as a `NULL` value in a MySQL database.

The `posix-time` sub-profile implementation also provides support for mapping `posix_time::ptime` to the `TIMESTAMP` MySQL type. However, this mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("TIMESTAMP") not_null
  boost::posix_time::ptime updated_;
};
```

Starting with MySQL version 5.6.4 it is possible to store fractional seconds up to microsecond precision in `TIME`, `DATETIME`, and `TIMESTAMP` columns. However, to enable sub-second precision, the corresponding type with the desired precision has to be specified explicitly, as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("DATETIME(6)")     // Microsecond precision.
  boost::posix_time::ptime updated_;
};
```

Alternatively, you can enable sub-second precision on the per-type basis, for example:

```
#pragma db value(boost::posix_time::ptime) type("DATETIME(6)")

#pragma db object
class person
{
  ...
  boost::posix_time::ptime created_; // Microsecond precision.
  boost::posix_time::ptime updated_; // Microsecond precision.
};
```

Some valid Boost date-time values cannot be stored in a MySQL database. An attempt to persist any Boost date-time special value other than `date_time::not_a_date_time` will result in the `special_value` exception. An attempt to persist a Boost date-time value that is out of the MySQL type range will result in the `out_of_range` exception. Refer to the MySQL documentation for more information on the MySQL data type ranges.

23.5.2 SQLite Database Type Mapping

The following table summarizes the default mapping between the currently supported Boost `date_time` types and the SQLite database types.

Boost `date_time` Type	SQLite Type	Default `NULL` Semantics
`gregorian::date`	`TEXT`	`NULL`
`posix_time::ptime`	`TEXT`	`NULL`
`posix_time::time_duration`	`TEXT`	`NULL`

The Boost special value `date_time::not_a_date_time` is stored as a `NULL` value in an SQLite database.

The `date-time` sub-profile implementation also provides support for mapping `gregorian::date` and `posix_time::ptime` to the `INTEGER` SQLite type, with the integer value representing the UNIX time. Similarly, an alternative mapping for `posix_time::time_duration` to the `INTEGER` type represents the duration as a number of seconds. These mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("INTEGER")
  boost::gregorian::date born_;
};
```

Some valid Boost date-time values cannot be stored in an SQLite database. An attempt to persist any Boost date-time special value other than `date_time::not_a_date_time` will result in the `special_value` exception. An attempt to persist a negative `posix_time::time_duration` value as SQLite `TEXT` will result in the `out_of_range` exception.

23.5.3 PostgreSQL Database Type Mapping

The following table summarizes the default mapping between the currently supported Boost `date_time` types and the PostgreSQL database types.

Boost `date_time` Type	PostgreSQL Type	Default `NULL` Semantics
`gregorian::date`	`DATE`	`NULL`
`posix_time::ptime`	`TIMESTAMP`	`NULL`
`posix_time::time_duration`	`TIME`	`NULL`

The Boost special value `date_time::not_a_date_time` is stored as a `NULL` value in a PostgreSQL database. `posix_time::ptime` values representing the special values `date_time::pos_infin` and `date_time::neg_infin` are stored as the special PostgreSQL TIMESTAMP values `infinity` and `-infinity`, respectively.

Some valid Boost date-time values cannot be stored in a PostgreSQL database. The PostgreSQL TIME type represents a clock time, and can therefore only store positive durations with a total length of time less than 24 hours. An attempt to persist a `posix_time::time_duration`

value outside of this range will result in the `value_out_of_range` exception. An attempt to persist a `posix_time::time_duration` value representing any special value other than `date_time::not_a_date_time` will result in the `special_value` exception.

23.5.4 Oracle Database Type Mapping

The following table summarizes the default mapping between the currently supported Boost `date_time` types and the Oracle database types.

Boost `date_time` Type	Oracle Type	Default `NULL` Semantics
`gregorian::date`	`DATE`	`NULL`
`posix_time::ptime`	`TIMESTAMP`	`NULL`
`posix_time::time_duration`	`INTERVAL DAY TO SECOND`	`NULL`

The Boost special value `date_time::not_a_date_time` is stored as a `NULL` value in an Oracle database.

The `date-time` sub-profile implementation also provides support for mapping `posix_time::ptime` to the `DATE` Oracle type with fractional seconds that may be stored in a `ptime` instance being ignored. This alternative mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("DATE")
  boost::posix_time::ptime updated_;
};
```

Some valid Boost date-time values cannot be stored in an Oracle database. An attempt to persist a `gregorian::date`, `posix_time::ptime`, or `posix_time::time_duration` value representing any special value other than `date_time::not_a_date_time` will result in the `special_value` exception.

23.5.5 SQL Server Database Type Mapping

The following table summarizes the default mapping between the currently supported Boost `date_time` types and the SQL Server database types.

Boost `date_time` Type	SQL Server Type	Default `NULL` Semantics
`gregorian::date`	`DATE`	`NULL`
`posix_time::ptime`	`DATETIME2`	`NULL`
`posix_time::time_duration`	`TIME`	`NULL`

The Boost special value `date_time::not_a_date_time` is stored as a `NULL` value in an SQL Server database.

Note that the `DATE`, `TIME`, and `DATETIME2` types are only available in SQL Server 2008 and later. SQL Server 2005 only supports the `DATETIME` and `SMALLDATETIME` date-time types. The new types are also unavailable when connecting to an SQL Server 2008 or later with the SQL Server 2005 Native Client ODBC driver.

The `date-time` sub-profile implementation provides support for mapping `posix_time::ptime` to the `DATETIME` and `SMALLDATETIME` types, however, this mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("DATETIME")
  boost::posix_time::ptime updated_;
};
```

Some valid Boost date-time values cannot be stored in an SQL Server database. An attempt to persist a `gregorian::date`, `posix_time::ptime`, or `posix_time::time_duration` value representing any special value other than `date_time::not_a_date_time` will result in the `special_value` exception. The range of the `TIME` type in SQL server is from `00:00:00.0000000` to `23:59:59.9999999`. An attempt to persist a `posix_time::time_duration` value out of this range will result in the `value_out_of_range` exception.

23.6 Uuid Library

The `uuid` sub-profile provides persistence support for the `uuid` type from the Boost `uuid` library. To enable only this profile, pass `boost/uuid` to the `--profile` ODB compiler option.

The manner in which these types are persisted is database system dependent and is discussed in the sub-sections that follow. By default a data member of the `uuid` type is mapped to a database column with `NULL` enabled and nil `uuid` instances are stored as a `NULL` value. However, you can change this behavior by declaring the data member `NOT NULL` with the `not_null` pragma (Section 14.4.6, "`null`/`not_null`"). In this case, or if the data member is an object id, the implementation will store nil `uuid` instances as zero UUID values (`{00000000-0000-0000-0000-000000000000}`). For example:

```
#pragma db object
class object
{
  ...

  boost::uuids::uuid x_; // Nil values stored as NULL.

  #pragma db not_null
  boost::uuids::uuid y_; // Nil values stored as zero.
};
```

23.6.1 MySQL Database Type Mapping

The following table summarizes the default mapping between the Boost `uuid` type and the MySQL database type.

Boost Type	MySQL Type	Default `NULL` Semantics
`boost::uuids::uuid`	`BINARY(16)`	`NULL`

23.6.2 SQLite Database Type Mapping

The following table summarizes the default mapping between the Boost `uuid` type and the SQLite database type.

Boost Type	SQLite Type	Default `NULL` Semantics
`boost::uuids::uuid`	`BLOB`	`NULL`

23.6.3 PostgreSQL Database Type Mapping

The following table summarizes the default mapping between the Boost `uuid` type and the PostgreSQL database type.

Boost Type	PostgreSQL Type	Default `NULL` Semantics
`boost::uuids::uuid`	`UUID`	`NULL`

23.6.4 Oracle Database Type Mapping

The following table summarizes the default mapping between the Boost `uuid` type and the Oracle database type.

Boost Type	Oracle Type	Default `NULL` Semantics
`boost::uuids::uuid`	`RAW(16)`	`NULL`

23.6.5 SQL Server Database Type Mapping

The following table summarizes the default mapping between the Boost `uuid` type and the SQL Server database type.

Boost Type	SQL Server Type	Default `NULL` Semantics
`boost::uuids::uuid`	`UNIQUEIDENTIFIER`	`NULL`

24 Qt Profile

The ODB profile implementation for Qt is provided by the `libodb-qt` library. Both Qt4 and Qt5 as well as C++98/03 and C++11 are supported.

The Qt profile consists of multiple sub-profiles corresponding to the common type groups within Qt. Currently, only types from the `QtCore` module are supported. To enable all the available Qt sub-profiles, pass `qt` as the profile name to the `--profile` ODB compiler option. Alternatively, you can enable only specific sub-profiles by passing individual sub-profile names to `--profile`. The following sections in this chapter discuss each Qt sub-profile in detail. The `qt` example in the `odb-examples` package shows how to enable and use the Qt profile.

Some sub-profiles may throw exceptions to indicate error conditions, such as the inability to store a specific value in a particular database system. All such exceptions derive from the `odb::qt::exception` class which in turn derives from the root of the ODB exception hierarchy, class `odb::exception` (Section 3.14, "ODB Exceptions"). The `odb::qt::exception` class is defined in the `<odb/qt/exception.hxx>` header file and has the same interface as `odb::exception`. Concrete exceptions that can be thrown by the Qt sub-profiles are described in the following sections.

24.1 Basic Types Library

The `basic` sub-profile provides persistence support for basic types defined by Qt. To enable only this profile, pass `qt/basic` to the `--profile` ODB compiler option.

The currently supported basic types are `QString`, `QByteArray`, and `QUuid`. The manner in which these types are persisted is database system dependent and is discussed in the sub-sections that follow. The example below shows how `QString` may be used within a persistent object.

```
#pragma db object
class Person
{
  ...
  QString name_;
};
```

By default a data member of the `QUuid` type is mapped to a database column with `NULL` enabled and null `QUuid` instances are stored as a `NULL` value. However, you can change this behavior by declaring the data member `NOT NULL` with the `not_null` pragma (Section 14.4.6, "`null`/`not_null`"). In this case, or if the data member is an object id, the implementation will store null `QUuid` instances as zero UUID values (`{00000000-0000-0000-0000-000000000000}`). For example:

```
#pragma db object
class object
{
  ...

  QUuid x_; // Null values stored as NULL.

  #pragma db not_null
  QUuid y_; // Null values stored as zero.
};
```

24.1.1 MySQL Database Type Mapping

The following table summarizes the default mapping between the currently supported basic Qt types and the MySQL database types.

Qt Type	MySQL Type	Default `NULL` Semantics
`QString`	`TEXT/VARCHAR(128)`	`NULL`
`QByteArray`	`BLOB`	`NULL`
`QUuid`	`BINARY(16)`	`NULL`

Instances of the `QString` and `QByteArray` types are stored as a `NULL` value if their `isNull()` member function returns `true`.

Note also that the `QString` type is mapped differently depending on whether a member of this type is an object id or not. If the member is an object id, then for this member `QString` is mapped to the `VARCHAR(128)` MySQL type. Otherwise, it is mapped to `TEXT`.

The `basic` sub-profile also provides support for mapping `QString` to the `CHAR`, `NCHAR`, and `NVARCHAR` MySQL types. However, these alternative mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "type"), as shown in the following example:

```
#pragma db object
class Person
{
  ...

  #pragma db type("CHAR(2)") not_null
  QString licenseState_;
};
```

24.1.2 SQLite Database Type Mapping

The following table summarizes the default mapping between the currently supported basic Qt types and the SQLite database types.

Qt Type	SQLite Type	Default `NULL` Semantics
`QString`	`TEXT`	`NULL`
`QByteArray`	`BLOB`	`NULL`
`QUuid`	`BLOB`	`NULL`

Instances of the `QString` and `QByteArray` types are stored as a `NULL` value if their `isNull()` member function returns `true`.

24.1.3 PostgreSQL Database Type Mapping

The following table summarizes the default mapping between the currently supported basic Qt types and the PostgreSQL database types.

Qt Type	PostgreSQL Type	Default `NULL` Semantics
`QString`	`TEXT`	`NULL`
`QByteArray`	`BYTEA`	`NULL`
`QUuid`	`UUID`	`NULL`

Instances of the `QString` and `QByteArray` types are stored as a `NULL` value if their `isNull()` member function returns `true`.

The `basic` sub-profile also provides support for mapping `QString` to the `CHAR` and `VARCHAR` PostgreSQL types. However, these alternative mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "type"), as shown in the following example:

```
#pragma db object
class Person
{
  ...

  #pragma db type("CHAR(2)") not_null
  QString licenseState_;
};
```

24.1.4 Oracle Database Type Mapping

The following table summarizes the default mapping between the currently supported basic Qt types and the Oracle database types.

Qt Type	Oracle Type	Default `NULL` Semantics
`QString`	`VARCHAR2(512)`	`NULL`
`QByteArray`	`BLOB`	`NULL`
`QUuid`	`RAW(16)`	`NULL`

Instances of the `QString` and `QByteArray` types are stored as a `NULL` value if their `isNull()` member function returns `true`.

The `basic` sub-profile also provides support for mapping `QString` to the `CHAR`, `NCHAR`, `NVARCHAR`, `CLOB`, and `NCLOB` Oracle types, and for mapping `QByteArray` to the `RAW` Oracle type. However, these alternative mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "type"), as shown in the following example:

```
#pragma db object
class Person
{
  ...

  #pragma db type("CLOB") not_null
  QString firstName_;

  #pragma db type("RAW(16)") null
  QByteArray uuid_;
};
```

24.1.5 SQL Server Database Type Mapping

The following table summarizes the default mapping between the currently supported basic Qt types and the SQL Server database types.

Qt Type	SQL Server Type	Default `NULL` Semantics
`QString`	`VARCHAR(512)/VARCHAR(256)`	`NULL`
`QByteArray`	`VARBINARY(max)`	`NULL`
`QUuid`	`UNIQUEIDENTIFIER`	`NULL`

Instances of the `QString` and `QByteArray` types are stored as a `NULL` value if their `isNull()` member function returns `true`.

Note also that the `QString` type is mapped differently depending on whether a member of this type is an object id or not. If the member is an object id, then for this member `QString` is mapped to the `VARCHAR(256)` SQL Server type. Otherwise, it is mapped to `VARCHAR(512)`.

The `basic` sub-profile also provides support for mapping `QString` to the `CHAR`, `NCHAR`, `NVARCHAR`, `TEXT`, and `NTEXT` SQL Server types, and for mapping `QByteArray` to the `BINARY` and `IMAGE` SQL Server types. However, these alternative mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "type"), as shown in the following example:

```
#pragma db object
class Person
{
  ...

  #pragma db type("NVARCHAR(256)") not_null
  QString firstName_;

  #pragma db type("BINARY(16)") null
  QByteArray uuid_;
};
```

24.2 Smart Pointers Library

The `smart-ptr` sub-profile provides persistence support the Qt smart pointers. To enable only this profile, pass `qt/smart-ptr` to the `--profile` ODB compiler option.

The currently supported smart pointers are `QSharedPointer` and `QWeakPointer`. For more information on using smart pointers as pointers to objects and views, refer to Section 3.3, "Object and View Pointers" and Chapter 6, "Relationships". For more information on using smart pointers as pointers to values, refer to Section 7.3, "Pointers and `NULL` Value Semantics". When used as a pointer to a value, only `QSharedPointer` is supported. For example:

```
#pragma db object
class person
{
  ...

  #pragma db null
  QSharedPointer<QString> middle_name_;
};
```

To provide finer grained control over object relationship loading, the `smart-ptr` sub-profile also provides the lazy counterparts for the above pointers: `QLazySharedPointer` and `QLazyWeakPointer`. You will need to include the `<odb/qt/lazy-ptr.hxx>` header file to make the lazy variants available in your application. For a description of the lazy pointer interface and semantics refer to Section 6.4, "Lazy Pointers". The following example shows how we can use these smart pointers to establish a relationship between persistent objects.

```
class Employee;

#pragma db object
class Position
{
  ...

  #pragma db inverse(position_)
  QLazyWeakPointer<Employee> employee_;
};

#pragma db object
class Employee
{
  ...

  #pragma db not_null
  QSharedPointer<Position> position_;
};
```

Besides providing persistence support for the above smart pointers, the `smart-ptr` sub-profile also changes the default pointer (Section 3.3, "Object and View Pointers") to `QShared-Pointer`. In particular, this means that database functions that return dynamically allocated objects and views will return them as `QSharedPointer` pointers. To override this behavior, add the `--default-pointer` option specifying the alternative pointer type after the `--profile` option.

24.3 Containers Library

The `containers` sub-profile provides persistence support for Qt containers. To enable only this profile, pass `qt/containers` to the `--profile` ODB compiler option.

The currently supported ordered containers are `QVector`, `QList`, and `QLinkedList`. Supported map containers are `QMap`, `QMultiMap`, `QHash`, and `QMultiHash`. The supported set container is `QSet`. For more information on using containers with ODB, refer to Chapter 5, "Containers". The following example shows how the `QSet` container may be used within a persistent object.

```
#pragma db object
class Person
{
  ...
  QSet<QString> emails_;
};
```

The `containers` sub-profile also provide a change-tracking equivalent for `QList` (Section 24.3.1, "Change-Tracking `QList`") with support for other Qt container equivalents planned for future releases. For general information on change-tracking containers refer to Section 5.4, "Change-Tracking Containers".

24.3.1 Change-Tracking `QList`

Class template `QOdbList`, defined in `<odb/qt/list.hxx>`, is a change-tracking equivalent for `QList`. It is implemented in terms of `QList` and is implicit-convertible to and implicit-constructible from `const QList&`. In particular, this means that we can use `QOdbList` instance anywhere `const QList&` is expected. In addition, `QOdbList` constant iterator (`const_iterator`) is the same type as that of `QList`.

`QOdbList` incurs 2-bit per element overhead in order to store the change state. It cannot be stored unordered in the database (Section 14.4.19 "`unordered`") but can be used as an inverse side of a relationship (6.2 "Bidirectional Relationships"). In this case, no change tracking is performed since no state for such a container is stored in the database.

The number of database operations required to update the state of `QOdbList` corresponds well to the complexity of `QList` functions, except for `prepend()`/`push_front()`. In particular, adding or removing an element from the back of the list (for example, with `append()`/`push_back()` and `removeLast()`/`pop_back()`), requires only a single database statement execution. In contrast, inserting or erasing an element at the beginning or in the middle of the list will require a database statement for every element that follows it.

`QOdbList` replicates most of the `QList` interface as defined in both Qt4 and Qt5 and includes support for C++11. However, functions and operators that provide direct write access to the elements had to be altered or disabled in order to support change tracking. Additional functions used to interface with `QList` and to control the change tracking state were also added. The following listing summarizes the differences between the `QOdbList` and `QList` interfaces. Any `QList` function or operator not mentioned in this listing has exactly the same signature and semantics in `QOdbList`. Functions and operators that were disabled are shown as commented out and are followed by functions/operators that replace them.

```
template <typename T>
class QOdbList
{
  ...
```

```
// Element access.
//

//T& operator[] (int);
  T& modify (int);

//T& first();
  T& modifyFirst();

//T& last();
  T& modifyLast();

//T& front();
  T& modify_front();

//T& back();
  T& modify_back();

// Iterators.
//
using const_iterator = typename QList<T>::const_iterator;

class iterator
{
  ...

  // Element Access.
  //

  //reference       operator* () const;
    const_reference operator* () const;
    reference       modify () const;

  //pointer       operator-> () const;
    const_pointer operator-> () const;

  //reference       operator[] (difference_type);
    const_reference operator[] (difference_type);
    reference       modify (difference_type) const;

  // Interfacing with QList::iterator.
  //
  typename QList<T>::iterator base () const;
};

// Return QList iterators. The begin() functions mark all
// the elements as modified.
//
typename QList<T>::iterator mbegin ();
typename QList<T>::iterator modifyBegin ();
```

```
  typename QList<T>::iterator mend ();
  typename QList<T>::iterator modifyEnd ();

  // Interfacing with QList.
  //
  QOdbList (const QList<T>&);
  QOdbList (QList<T>&&); // C++11 only.

  QOdbList& operator= (const QList<T>&);
  QOdbList& operator= (QList<T>&&);

  operator const QList<T>& () const;
  QList<T>& base ();
  const QList<T>& base () const;

  // Change tracking.
  //
  bool _tracking () const;
  void _start () const;
  void _stop () const;
  void _arm (transaction&) const;
};
```

The following example highlights some of the differences between the two interfaces. `QList` versions are commented out.

```
#include <QtCore/QList>
#include <odb/qt/list.hxx>

void f (const QList<int>&);

QOdbList<int> l ({1, 2, 3});

f (l); // Ok, implicit conversion.

if (l[1] == 2) // Ok, const access.
  //l[1]++;
  l.modify (1)++;

//l.last () = 4;
l.modifyLast () = 4;

for (auto i (l.begin ()); i != l.end (); ++i)
{
  if (*i != 0) // Ok, const access.
    //*i += 10;
    i.modify () += 10;
}

qSort (l.modifyBegin (), l.modifyEnd ());
```

Note also the subtle difference between copy/move construction and copy/move assignment of `QOdbList` instances. While copy/move constructor will copy/move both the elements as well as their change state, in contrast, assignment is tracked as any other change to the vector content.

The `QListIterator` and `QMutableListIterator` equivalents are also provided. These are `QOdbListIterator` and `QMutableOdbListIterator` and are defined in `<odb/qt/list-iterator.hxx>` and `<odb/qt/mutable-list-iterator.hxx>`, respectively.

`QOdbListIterator` has exactly the same interface and semantics as `QListIterator`. In fact, we can use `QListIterator` to iterate over a `QOdbList` instance.

`QMutableOdbListIterator` also has exactly the same interface as `QMutableListIterator`. Note, however, that any element that such an iterator passes over with the call to `next()` is marked as modified.

24.4 Date Time Library

The `date-time` sub-profile provides persistence support for the Qt date-time types. To enable only this profile, pass `qt/date-time` to the `--profile` ODB compiler option.

The currently supported date-time types are `QDate`, `QTime`, and `QDateTime`. The manner in which these types are persisted is database system dependent and is discussed in the sub-sections that follow. The example below shows how `QDate` may be used within a persistent object.

```
#pragma db object
class Person
{
  ...
  QDate dateOfBirth_;
};
```

The single concrete exception that can be thrown by the `date-time` sub-profile implementation is presented below.

```
namespace odb
{
  namespace qt
  {
    namespace date_time
    {
      struct value_out_of_range: odb::qt::exception
      {
        virtual const char*
        what () const throw ();
```

```
      };
    }
  }
}
```

You will need to include the `<odb/qt/date-time/exceptions.hxx>` header file to make this exception available in your application.

The `value_out_of_range` exception is thrown if an attempt is made to store a date-time value that is out of the target database range. The specific conditions under which it is thrown is database system dependent and is discussed in more detail in the following sub-sections.

24.4.1 MySQL Database Type Mapping

The following table summarizes the default mapping between the currently supported Qt date-time types and the MySQL database types.

Qt Date Time Type	MySQL Type	Default `NULL` Semantics
`QDate`	`DATE`	`NULL`
`QTime`	`TIME`	`NULL`
`QDateTime`	`DATETIME`	`NULL`

Instances of the `QDate`, `QTime`, and `QDateTime` types are stored as a `NULL` value if their `isNull()` member function returns true.

The `date-time` sub-profile implementation also provides support for mapping `QDateTime` to the `TIMESTAMP` MySQL type. However, this mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class Person
{
  ...
  #pragma db type("TIMESTAMP") not_null
  QDateTime updated_;
};
```

Starting with MySQL version 5.6.4 it is possible to store fractional seconds up to microsecond precision in `TIME`, `DATETIME`, and `TIMESTAMP` columns. However, to enable sub-second precision, the corresponding type with the desired precision has to be specified explicitly, as shown in the following example:

```
#pragma db object
class Person
{
  ...
  #pragma db type("DATETIME(3)") // Millisecond precision.
  QDateTime updated_;
};
```

Alternatively, you can enable sub-second precision on the per-type basis, for example:

```
#pragma db value(QDateTime) type("DATETIME(3)")

#pragma db object
class Person
{
  ...
  QDateTime created_; // Millisecond precision.
  QDateTime updated_; // Millisecond precision.
};
```

Some valid Qt date-time values cannot be stored in a MySQL database. An attempt to persist a Qt date-time value that is out of the MySQL type range will result in the `out_of_range` exception. Refer to the MySQL documentation for more information on the MySQL data type ranges.

24.4.2 SQLite Database Type Mapping

The following table summarizes the default mapping between the currently supported Qt date-time types and the SQLite database types.

Qt Date Time Type	SQLite Type	Default `NULL` Semantics
`QDate`	`TEXT`	`NULL`
`QTime`	`TEXT`	`NULL`
`QDateTime`	`TEXT`	`NULL`

Instances of the `QDate`, `QTime`, and `QDateTime` types are stored as a `NULL` value if their `isNull()` member function returns true.

The `date-time` sub-profile implementation also provides support for mapping `QDate` and `QDateTime` to the SQLite `INTEGER` type, with the integer value representing the UNIX time. Similarly, an alternative mapping for `QTime` to the `INTEGER` type represents a clock time as the number of seconds since midnight. These mappings have to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class Person
{
  ...
  #pragma db type("INTEGER")
  QDate born_;
};
```

Some valid Qt date-time values cannot be stored in an SQLite database. An attempt to persist any Qt date-time value representing a negative UNIX time (any point in time prior to the 1970-01-01 00:00:00 UNIX time epoch) as an SQLite `INTEGER` will result in the `out_of_range` exception.

24.4.3 PostgreSQL Database Type Mapping

The following table summarizes the default mapping between the currently supported Qt date-time types and the PostgreSQL database types.

Qt Date Time Type	PostgreSQL Type	Default `NULL` Semantics
`QDate`	`DATE`	`NULL`
`QTime`	`TIME`	`NULL`
`QDateTime`	`TIMESTAMP`	`NULL`

Instances of the `QDate`, `QTime`, and `QDateTime` types are stored as a `NULL` value if their `isNull()` member function returns true.

24.4.4 Oracle Database Type Mapping

The following table summarizes the default mapping between the currently supported Qt date-time types and the Oracle database types.

Qt Date Time Type	Oracle Type	Default `NULL` Semantics
`QDate`	`DATE`	`NULL`
`QTime`	`INTERVAL DAY(0) TO SECOND(3)`	`NULL`
`QDateTime`	`TIMESTAMP(3)`	`NULL`

Instances of the `QDate`, `QTime`, and `QDateTime` types are stored as a `NULL` value if their `isNull()` member function returns true.

The `date-time` sub-profile implementation also provides support for mapping `QDateTime` to the `DATE` Oracle type with fractional seconds that may be stored in a `QDateTime` instance being ignored. This alternative mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("DATE")
  QDateTime updated_;
};
```

24.4.5 SQL Server Database Type Mapping

The following table summarizes the default mapping between the currently supported Qt date-time types and the SQL Server database types.

Qt Date Time Type	SQL Server Type	Default `NULL` Semantics
`QDate`	`DATE`	`NULL`
`QTime`	`TIME(3)`	`NULL`
`QDateTime`	`DATETIME2(3)`	`NULL`

Instances of the `QDate`, `QTime`, and `QDateTime` types are stored as a `NULL` value if their `isNull()` member function returns true.

Note that the `DATE`, `TIME`, and `DATETIME2` types are only available in SQL Server 2008 and later. SQL Server 2005 only supports the `DATETIME` and `SMALLDATETIME` date-time types. The new types are also unavailable when connecting to an SQL Server 2008 or later with the SQL Server 2005 Native Client ODBC driver.

The `date-time` sub-profile implementation provides support for mapping `QDateTime` to the `DATETIME` and `SMALLDATETIME` types, however, this mapping has to be explicitly requested using the `db type` pragma (Section 14.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("DATETIME")
  QDateTime updated_;
};
```