C++ Object Persistence with ODB

Revision 1.7, December 2011

This revision of the manual describes ODB 1.7.0 and is available in the following formats: XHTML, PDF, and PostScript.

Table of Contents

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

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, and run native SQL `SELECT` queries. 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 conclude the chapter by contrasting the drawbacks of the traditional way of saving C++ objects to relational databases with the benefits of using ODB for object persistence. 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

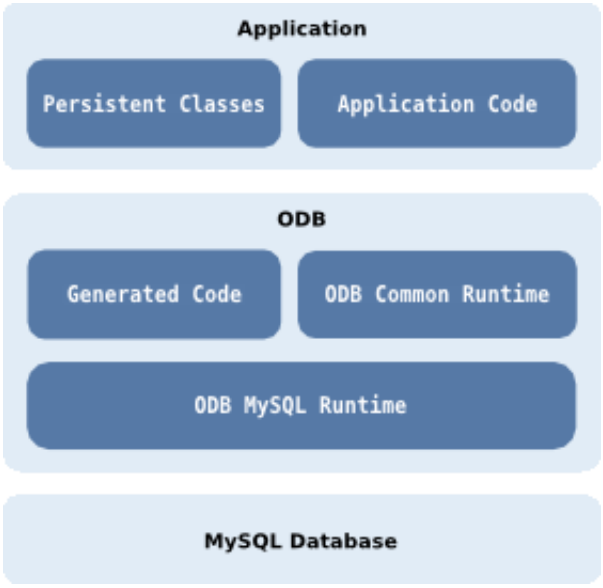
From the application developer’s perspective, ODB consists of three main components: the ODB compiler, the common runtime library, called `libodb`, and the database-specific runtime libraries, called `libodb-<database>`, where <database> is the name of the database system this runtime is for, for example, `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:

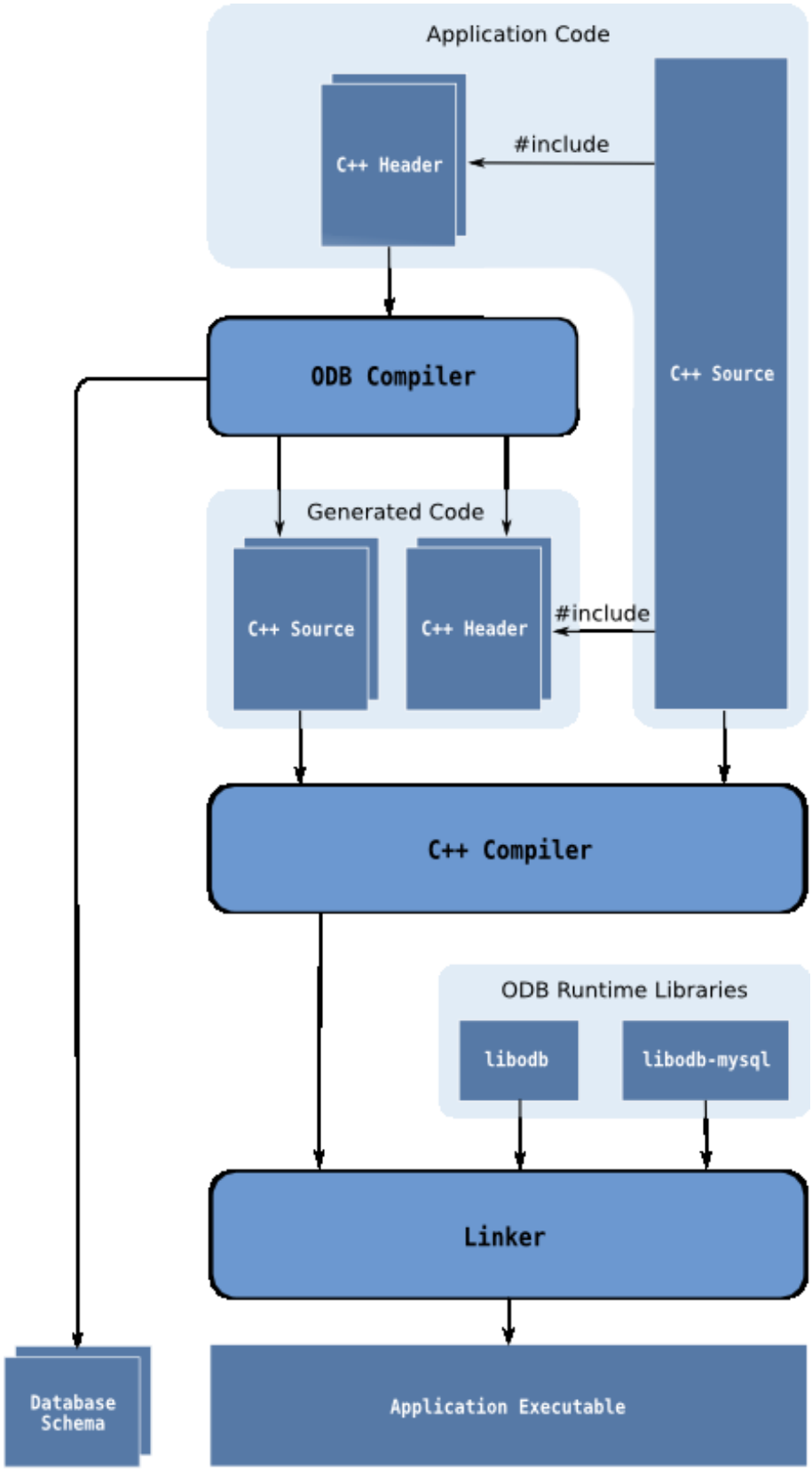


The ODB system also defines two special-purpose languages: the ODB Pragma Language and ODB Query Language. The ODB Pragma Language is used to communicate various properties of persistent classes to the ODB compiler by means of special `#pragma` directives embedded in the C++ header files. It controls 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.

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.

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 a Persistent Class

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 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.

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 ODB support code. If your class has public default constructor and public 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 must have a unique, within its class, identifier. Or, in other words, no two persistent instances of the same type have equal identifiers. 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 since each person is presumed to have a unique email address, for example:

```
class person
{
  ...

  #pragma db id
  std::string email_;

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

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 3.3, "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::auto_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
  {
    auto_ptr<database> db (new odb::mysql::database (argc, argv));

    unsigned 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 information directly (Section 13.2, "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 3.13, "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 3.12, "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.
    //
    {
      ...
    }

    typedef odb::query<person> query;
    typedef odb::result<person> result;

    // 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 `typedef`s 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 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 ());
```

```
      auto_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::auto_ptr` to manage the returned object, we could have also used another smart pointer, for example `shared_ptr` from TR1 or Boost. For more information on the object lifetime management and the smart pointers that we can use for that, see Section 3.2, "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 ());

  result r (db->query<person> (query::first == "Joe" &&
                               query::last == "Dirt"));

  result::iterator i (r.begin ());

  if (i != r.end ())
  {
    auto_ptr<person> joe (i.load ());
    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.

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 9, "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;
};
```

To get the result of a view we use the same `query()` function as when querying the database for an object. 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 ());

  odb::result<person_stat> r (db->query<person_stat> ());

  // The result of this query always has exactly one element.
  //
  const person_stat& ps (*r.begin ());

  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 ());

  result r (db->query<person> (query::first == "John" &&
                               query::last == "Doe"));

  result::iterator i (r.begin ());

  if (i != r.end ())
  {
    auto_ptr<person> john (i.load ());
    db->erase (*john);
  }

  t.commit ();
}
```

2.9 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 3.2 and continue with the discussion of the `odb::database` class in Section 3.3, transactions in Section 3.4, and connections in Section 3.5. 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.

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 9, "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.

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 id_;
};
```

While it is possible to declare a persistent class without an object id, such a class will have limited functionality (Section 12.1.6, "`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 12, "ODB Pragma Language").

Normally, an object 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. It is also possible to have an object type without the default constructor. However, in this case, the database operations can only load the persistent state into an existing instance (Section 3.8, "Loading Persistent Objects", Section 4.4, "Query Result").

The object id type should be default-constructible.

If an object class has private or protected non-transient data members or if its default constructor is not public, then the `odb::access` class, defined in the `<odb/core.hxx>` header, should be declared a friend of this object type. For example:

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

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

```
private:
  friend class odb::access;

  person () {}

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

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 system 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 system 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 12.3.1, "`type`". Similar to object types, 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_;
};
```

Composite value types are discussed in more detail in Section 7.2, "Composite Value Types".

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 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 10, "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` or `shared_ptr` from TR1 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 and view pointers. The `shared_ptr` smart pointer from TR1 or Boost is a good default choice.

ODB provides two mechanisms for changing the object or view pointer type. We can use the `--default-pointer` option to specify the default pointer. All objects and views that don't have the pointer type explicitly specified with the `db pointer` pragma (see below) will use the default pointer type. Refer to the ODB Compiler Command Line Manual for details on this option's argument. The typical usage is shown below:

```
--default-pointer std::tr1::shared_ptr
```

The second mechanism allows us to specify the pointer type on the per object and per view basis using the `db pointer` pragma, for example:

```
#pragma db object pointer(std::tr1::shared_ptr)
class person
{
  ...
};
```

Refer to Section 12.1.2, "`pointer` (object)" and Section 12.2.4, "`pointer` (view)" for more information on this pragma.

Built-in support that is provided by the ODB runtime library allows us to use the TR1 `shared_ptr` and `std::auto_ptr` 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 6.4, "Using Custom Smart Pointers".

3.3 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>

auto_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 either as a standalone SQL file or embedded into the generated C++ code (`--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, the schema can be embedded directly into the generated code and 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 = "");
  };
}
```

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 `--default-schema` ODB compiler option to assign custom schema names and then use these names as a second argument to `create_schema()`. If the schema is not found, `create_schema()` throws the `odb::unknown_schema` exception. The `create_schema()` function should be called within a transaction.

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 12, "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.

3.4 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:
    typedef odb::database database_type;
    typedef odb::connection connection_type;

    transaction (transaction_impl*, bool make_current = true)

    void
    commit ();

    void
    rollback ();

    database_type&
    database ();

    connection_type&
    connection ();

    static bool
    has_current ();

    static transaction&
    current ();

    static void
    current (transaction&);

    static bool
    reset_current ();
  };
}
```

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 `odb::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 3.5, "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 ();
```

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 id)
{
  transaction t (db.begin ());

  auto_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;
  }
}
```

3.5 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:
    typedef odb::database database_type;

    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 ();
  };

  typedef details::shared_ptr<connection> connection_ptr;
}
```

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 3.11, "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.

3.6 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 3.13, "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;
  }
}
```

3.7 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 3.2, "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 12.4.2, "`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 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.

3.8 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 ());

auto_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 10.1, "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 11, "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.

3.9 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 from_acc,
          unsigned 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 11, "Optimistic Concurrency".

In ODB, persistent classes, composite value types, as well as individual data members can be declared read-only (see Section 12.1.4, "`readonly` (object)", Section 12.3.6, "`readonly` (composite value)", and Section 12.4.10, "`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.

3.10 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 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 11, "Optimistic Concurrency".

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:

```
typedef odb::query<person> query;

transaction t (db.begin ());

db.erase_query<person> (query::last == "Doe" && query::are < 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.

```
typedef odb::query<employee> query;

transaction t (db.begin ());

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

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

t.commit ();
```

3.11 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 3.5, "Connections".

3.12 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` is a built-in tracer implementation provided by the ODB runtime. It prints each executed SQL statement to the standard error stream.

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.
```

3.13 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 ();
};

// 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
  {
  };

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

    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 3.4, "Transactions".

The next two exceptions (`already_in_session`, and `not_in_session`) are thrown by the `odb::session` class and are discussed in Chapter 10, "Session".

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 3.6, "Error Handling and Recovery" for details.

The `object_already_persistent` exception is thrown by the `persist()` database function. See Section 3.7, "Making Objects Persistent" for details.

The `object_not_persistent` exception is thrown by the `load()`, `update()`, and `erase()` database functions. Refer to Section 3.8, "Loading Persistent Objects", Section 3.9, "Updating Persistent Objects", and Section 3.10, "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 11, "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 `unknown_schema` exception is thrown by the `odb::schema_catalog` class if a schema with the specified name is not found. Refer to Section 3.3, "Database" 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:

```
typedef odb::query<person> query;
typedef odb::result<person> result;

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'" + (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:

```
typedef odb::query<person> query;

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)`
`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 ()));
```

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:

```
typedef odb::query<person> query;
typedef odb::result<person> result;

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. 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_underage (db, query::first == "John"));
```

4.4 Query Result

The result of executing a query is zero, one, or more objects matching the query criteria. The result is returned as an instance of the `odb::result` class template, for example:

```
  typedef odb::query<person> query;
  typedef odb::result<person> result;

  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 file 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:
    typedef odb::result_iterator<T> iterator;

  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.

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)
{
  ...
}
```

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.

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")
  {
    auto_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> 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.
  }
}
```

5 Containers

The ODB runtime library provides built-in persistence support for all the commonly used standard C++ containers, namely, `std::vector`, `std::list`, `std::set`, `std::multiset`, `std::map`, and `std::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"). It is also easy to persist custom container types as discussed later in Section 5.4, "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` and `std::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.

Consider the following persistent object as an example:

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

```
  #pragma db id auto
  unsigned 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 12, "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 12.3.7, "`unordered`", Section 12.4.13, "`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::multi-set`.

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.

Consider the following persistent object as an example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id auto
  unsigned 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 12, "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`.

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.

Consider the following persistent object as an example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id auto
  unsigned 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 12, "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 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.

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 the TR1 `shared_ptr`/`weak_ptr`, `std::auto_ptr`, and raw pointers. Plus, ODB profile libraries, that 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 6.4, "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 3.2, "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 6.3, "Lazy Pointers".

As a simple example, consider the following employee-employer relationship. Code examples presented in this chapter will use the `shared_ptr` and `weak_ptr` smart pointers from the TR1 (`std::tr1`) namespace.

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

  #pragma db id
  std::string name_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned 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 12.4.4, "`null`/`not_null`") for single object pointers and the `value_not_null` pragma (Section 12.4.18, "`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 perform a database operation 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 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 10, "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:

```
typedef odb::query<employee> query;
typedef odb::result<employee> result;

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.

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 id_;

  #pragma db not_null
  shared_ptr<employer> employer_;
};
```

The corresponding database tables look like this:

```
CREATE TABLE employer (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employer VARCHAR (255) 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 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 (255) 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 (255) 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 12, "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 (255) 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 id_;

  weak_ptr<employee> employee_;
};

#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned 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_;
};
```

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 12.4.11, "`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 12.4.13, "`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 id_;

  #pragma db inverse(position_)
  weak_ptr<employee> employee_;
};
```

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

  #pragma db id
  unsigned 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 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 (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employer VARCHAR (255) 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 (255) NOT NULL PRIMARY KEY);

CREATE TABLE employer_employees (
  object_id VARCHAR (255) NOT NULL,
  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 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 (255) 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 (255) 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 (255) NOT NULL PRIMARY KEY);

CREATE TABLE project_employees (
  object_id VARCHAR (255) NOT NULL,
  value BIGINT UNSIGNED NOT NULL REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);
```

6.3 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 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 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. A 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` and `odb::lazy_weak_ptr` for TR1 `shared_ptr` and `weak_ptr`, `odb::lazy_auto_ptr` for `std::auto_ptr`, and `odb::lazy_ptr` for raw pointers. The ODB profile libraries 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 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.
  //
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;

  // 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 */>
  object_traits<O>::id_type object_id () const;

  odb::database& database () const;
};
```

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.

```
typedef std::vector<lazy_weak_ptr<employee> > employees;

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)
  {
    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 ();
```

6.4 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 12.3.1, "`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 chapter 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 which case the default constructor can be made private. If a composite value type has private or protected non-transient data members or if its default constructor is not public and the value type is used as an element of a container, then the `odb::access` class should be declared a friend of this value type. For example:

```
#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.

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

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_;
};

typedef std::vector<basic_name> basic_names;

#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_;
};
```

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:

```
typedef odb::query<person> query;
typedef odb::result<person> result;

transaction t (db->begin ());

result r (db->query<person> (
  query::name.extras.nickname == "Squeaky"));

...

t.commit ();
```

7.2.1 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 12.4.7, "`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 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 12.4.26, "`value_column`") or `db key_column` (Section 12.4.25, "`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 12.4.14, "`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 12.4.4, "`null`/`not_null`").

To properly support the `NULL` semantics, the C++ value type must have a notion or 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:
    typedef T value_type;

    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 18.3, "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::auto_ptr<std::string> middle_;

  std::string last_;
};
```

The ODB compiler includes built-in support for using `std::auto_ptr` and `std::tr1::shared_ptr` 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").

Currently, ODB supports the `NULL` semantics only for simple values. In future versions this support will be extended to composite values and containers. With this limitation in mind, we can still use smart pointers in data members of composite value and container types. The only restriction is that these pointers must not be `NULL`. For example:

```
#pragma db value
struct name
{
  std::string first_;
  std::string last_;
};

#pragma db object
class person
{
```

```
  ...

  std::auto_ptr<name> name_;
  std::auto_ptr<std::vector<name> > 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 derived classes instead of a 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 12.1.3, "`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 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 (255) NOT NULL PRIMARY KEY);
```

The complete version of the code presented in this section is available in the `inheritance` example in the `odb-examples` package.

8.2 Polymorphism Inheritance

Polymorphism inheritance mapping is not yet implemented. Future versions of ODB will add support for this functionality.

9 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, 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 id_;

  std::string name_;
};

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

```
  #pragma db id
  unsigned 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()` function in exactly the same way as we would use it 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:

```
typedef odb::query<employee_name> query;
typedef odb::result<employee_name> result;

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.

9.1 Object Views

To associate one or more objects with a view we use the `db object` pragma (Section 12.2.1, "`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-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-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 9.3, "Mixed Views", tables. Note that while the first associated object can have an alias, it cannot have a join 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 12.4.7, "`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 12.4.3, "`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:

```
typedef odb::result<employee_count> result;
typedef odb::query<employee_count> query;

transaction t (db.begin ());

// Find the number of employees with the Doe last name.
//
result r (db.query<employee_count> (query::last == "Doe"));

// Result of this aggregate query contains only one element.
//
cout << r.begin ()->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:

```
typedef odb::result<employee_employer> result;
typedef odb::query<employee_employer> query;

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:

```
typedef odb::query<employee_country> query;
typedef odb::result<employee_country> result;

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:

```
typedef odb::query<employee_name> query;
typedef odb::result<employee_name> result;

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.

9.2 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 12.2.2, "`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 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;
};
```

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-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-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 9.3, "Mixed Views", objects. Note that while the first associated table can have an alias, it cannot have a join 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 corre-

sponding 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 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:

```
typedef odb::query<employee_leave> query;
typedef odb::result<employee_leave> result;

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 ();
```

9.3 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:

```
typedef odb::query<employee_vacation> query;
typedef odb::result<employee_vacation> result;

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;
};
```

9.4 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 12.2.3, "`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 a 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;
};
```

9.5 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. Native views don't have associated tables or objects. Instead, we use the `db query` pragma to specify the native SQL query, which must at a minimum 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 9.2 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 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 automatically 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:

```
typedef odb::query<sequence_value> query;
typedef odb::result<sequence_value> result;

string seq_name = ...

result l (db.query<sequence_value> (
  "SELECT lastval('" + seq_name + "')"));

result n (db.query<sequence_value> (
  "SELECT nextval('" + seq_name + "')"));
```

9.6 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;
};
```

10 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 will provide additional functionality, such as automatic object state change tracking.

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 ();
    ~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 ();

    // Object cache interface.
    //
  public:
    typedef odb::database database_type;

    template <typename T>
    void
    insert (database_type&,
            const object_traits<T>::id_type&,
            const object_traits<T>::pointer_type&);

    template <typename T>
    object_traits<T>::pointer_type
    find (database_type&, const object_traits<T>::id_type&) const;

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

The session constructor creates a new session and sets it as a current session for this thread. If we try to create another session while there is already a current session in effect, 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.

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.

10.1 Object Cache

A session is an object cache. Every time an object is made persistent by calling the `database::persist()` function (Section 3.7, "Making Objects Persistent"), loaded by calling the `database::load()` or `database::find()` function (Section 3.8, "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 3.2, "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 3.10, "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 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 6.4, "Using Custom Smart Pointers"). Objects with a unique pointer, such as `std::auto_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 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 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 Optimistic Concurrency

The ODB transaction model (Section 3.4, "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 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 12.1.5, "`optimistic`"). We also use the `version` pragma (Section 12.4.12, "`version`") to specify which data member will store the object version. For example:

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

  #pragma db version
  unsigned 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 version_;
};
```

When we call the `database::update()` function (Section 3.9, "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 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 3.10, "Deleting Persistent Objects"). To understand why this can be important, consider the following application transaction:

```
unsigned 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 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.

12 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`, and `member`. 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 *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 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 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 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 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 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 three sections cover the specifiers applicable to the `object`, `value`, and `member` qualifiers.

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 12.5, "C++ Compiler Warnings".

12.1 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	12.1.1
`pointer`	pointer type for a persistent class	12.1.2
`abstract`	persistent class is abstract	12.1.3
`readonly`	persistent class is read-only	12.1.4
`optimistic`	persistent class with the optimistic concurrency model	12.1.5
`id`	persistent class has no object id	12.1.6
`callback`	database operations callback	12.1.7

12.1.1 `table`

The `table` specifier specifies the table name that should be used to store objects of a 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.

12.1.2 `pointer`

The `pointer` specifier specifies the object pointer type for a 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::tr1::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::tr1::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 with the `--default-pointer` ODB compiler option, is used. If this option is not specified either, then the raw pointer is used by default.

For a more detailed discussion of object pointers, refer to Section 3.2, "Object and View Pointers".

12.1.3 `abstract`

The `abstract` specifier specifies that a 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".

12.1.4 `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 3.9, "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 12.4.10, "`readonly`") as well as composite value types (Section 12.3.6, "`readonly`") as read-only.

12.1.5 `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 12.4.12, "`version`"). For example:

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

  #pragma db version
  unsigned 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 11, "Optimistic Concurrency".

12.1.6 `id`

The `id` specifier specifies that the persistent class has no object id. It should be followed by opening and closing parenthesis. For example:

```
#pragma db object 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 3.8, "Loading Persistent Objects"), updated with the `database::update()` function (Section 3.9, "Updating Persistent Objects"), or deleted with the `database::erase()` function (Section 3.10, "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 3.10, "Deleting Persistent Objects") functions, respectively. There is no way to update such objects except by using native SQL statements (Section 3.11, "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 10.1, "Object Cache") either.

To declare a persistent class with an object id, use the data member `id` specifier (Section 12.4.1, "`id`").

12.1.7 `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.

If only the non-`const` version of the callback function is provided, then only database operations that are performed on unrestricted objects will trigger callback calls. If only the `const` version is provided, then the database operations on both constant and unrestricted objects will trigger callback calls but the object will always be passed as constant. Finally, if both versions are provided, then the `const` overload will be called for constant objects and the non-`const` overload for unrestricted objects. These rules are modeled after the standard C++ overload resolution rules. A callback function can be inline or virtual.

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;
    }
  }
};
```

12.2 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	12.2.1
`table`	table associated with a view	12.2.2
`query`	view query condition	12.2.3
`pointer`	pointer type for a view	12.2.4
`callback`	database operations callback	12.2.5

For more information on view types refer to Chapter 9, "Views".

12.2.1 `object`

The `object` specifier specifies a persistent class that should be associated with a view. For more information on object associations refer to Section 9.1, "Object Views".

12.2.2 `table`

The `table` specifier specifies a database table that should be associated with a view. For more information on table associations refer to Section 9.2, "Table Views".

12.2.3 `query`

The `query` specifier specifies a query condition 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 9.4, "View Query Conditions". For more information on native SQL queries, refer to Section 9.5, "Native Views".

12.2.4 `pointer`

The `pointer` specifier specifies the view pointer type for a 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 12.1.2, "`pointer`").

12.2.5 `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 12.1.7, "`callback`") except that the only events that can trigger a callback call in the case of a view are `pre_load` and `post_load`.

12.3 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	12.3.1
`id_type`	database type for a value type when used as an object id	12.3.2
`null`/`not_null`	type can/cannot be `NULL`	12.3.3
`default`	default value for a value type	12.3.4
`options`	database options for a value type	12.3.5
`readonly`	composite value type is read-only	12.3.6
`unordered`	ordered container should be stored unordered	12.3.7
`index_type`	database type for a container's index type	12.3.8
`key_type`	database type for a container's key type	12.3.9
`value_type`	database type for a container's value type	12.3.10
`value_null`/`value_not_null`	container's value can/cannot be `NULL`	12.3.11
`id_options`	database options for a container's id column	12.3.12
`index_options`	database options for a container's index column	12.3.13
`key_options`	database options for a container's key column	12.3.14
`value_options`	database options for a container's value column	12.3.15
`id_column`	column name for a container's object id	12.3.16
`index_column`	column name for a container's index	12.3.17
`key_column`	column name for a container's key	12.3.18
`value_column`	column name for a container's value	12.3.19

Many of the value type specifiers have corresponding member type specifiers with the same names (Section 12.4, "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 speci-

fier, 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.

12.3.1 `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 12.3.3, "`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.

12.3.2 `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 12.4.1, "`id`"). In combination with the `type` specifier (Section 12.3.1, "`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(128)")

#pragma db object
class person
{
  ...

  #pragma db id
  std::string email_; // Mapped to VARCHAR(128) 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_;
};
```

12.3.3 `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 std::tr1::shared_ptr;

typedef shared_ptr<std::string> string_ptr;
#pragma db value(string_ptr) type("TEXT") null

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

```
  string_ptr name_; // Mapped to TEXT NULL.
};

typedef shared_ptr<person> person_ptr;
#pragma db value(person_ptr) not_null
```

The `NULL` semantics can also be specified on the per-member basis (Section 12.4.4, "`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".

12.3.4 `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 12.4.5, "`default`").

12.3.5 `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 12.4.6, "`options`").

12.3.6 `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 3.9, "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 12.4.10, "`readonly`") as well as whole objects (Section 12.1.4, "`readonly`") as read-only.

12.3.7 `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:

```
typedef std::vector<std::string> names;
#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".

12.3.8 `index_type`

The `index_type` specifier specifies the native database type that should be used for an ordered container's index column. The semantics of `index_type` are similar to those of the `type` specifier (Section 12.3.1, "`type`"). The native database type is expected to be an integer type. For example:

```
typedef std::vector<std::string> names;
#pragma db value(names) index_type("SMALLINT UNSIGNED")
```

12.3.9 `key_type`

The `key_type` specifier specifies the native database type that should be used for a map container's key column. The semantics of `key_type` are similar to those of the `type` specifier (Section 12.3.1, "`type`"). For example:

```
typedef std::map<unsigned short, float> age_weight_map;
#pragma db value(age_weight_map) key_type("INT UNSIGNED")
```

12.3.10 `value_type`

The `value_type` specifier specifies the native database type that should be used for a container's value column. The semantics of `value_type` are similar to those of the `type` specifier (Section 12.3.1, "`type`"). For example:

```
typedef std::vector<std::string> names;
#pragma db value(names) value_type("VARCHAR(255)")
```

The `value_null` and `value_not_null` (Section 12.3.11, "`value_null`/`value_not_null`") specifiers can be used to control the NULL semantics of a value column.

12.3.11 `value_null`/`value_not_null`

The `value_null` and `value_not_null` specifiers specify that a 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 12.3.3, "`null`/`not_null`"). For example:

```
using std::tr1::shared_ptr;

#pragma db object
class account
{
  ...
};

typedef std::vector<shared_ptr<account> > accounts;
#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.

12.3.12 `id_options`

The `id_options` specifier specifies additional column definition options that should be used for a container's id column. For example:

```
typedef std::vector<std::string> nicknames;
#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 12.4.19, "`id_options`").

12.3.13 `index_options`

The `index_options` specifier specifies additional column definition options that should be used for a container's index column. For example:

```
typedef std::vector<std::string> nicknames;
#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 12.4.20, "`index_options`").

12.3.14 `key_options`

The `key_options` specifier specifies additional column definition options that should be used for a container's key column. For example:

```
typedef std::map<std::string, std::string> properties;
#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 12.4.21, "`key_options`").

12.3.15 `value_options`

The `value_options` specifier specifies additional column definition options that should be used for a container's value column. For example:

```
typedef std::set<std::string> nicknames;
#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 12.4.22, "`value_options`").

12.3.16 `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 example:

```
typedef std::vector<std::string> names;
#pragma db value(names) id_column("id")
```

If the column name is not specified, then `object_id` is used by default.

12.3.17 `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 example:

```
typedef std::vector<std::string> names;
#pragma db value(names) index_column("name_number")
```

If the column name is not specified, then `index` is used by default.

12.3.18 `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 example:

```
typedef std::map<unsigned short, float> age_weight_map;
#pragma db value(age_weight_map) key_column("age")
```

If the column name is not specified, then `key` is used by default.

12.3.19 `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 example:

```
typedef std::map<unsigned short, float> age_weight_map;
#pragma db value(age_weight_map) value_column("weight")
```

If the column name is not specified, then `value` is used by default.

12.4 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	12.4.1
`auto`	id is assigned by the database	12.4.2
`type`	database type for a member	12.4.3
`null`/`not_null`	member can/cannot be `NULL`	12.4.4
`default`	default value for a member	12.4.5
`options`	database options for a member	12.4.6
`column`	column name for a member of an object or composite value	12.4.7
`column`	column name for a member of a view	12.4.8
`transient`	member is not stored in the database	12.4.9

`readonly`	member is read-only	12.4.10
`inverse`	member is an inverse side of a bidirectional relationship	12.4.11
`version`	member stores object version	12.4.12
`unordered`	ordered container should be stored unordered	12.4.13
`table`	table name for a container	12.4.14
`index_type`	database type for a container's index type	12.4.15
`key_type`	database type for a container's key type	12.4.16
`value_type`	database type for a container's value type	12.4.17
`value_null`/`value_not_null`	container's value can/cannot be `NULL`	12.4.18
`id_options`	database options for a container's id column	12.4.19
`index_options`	database options for a container's index column	12.4.20
`key_options`	database options for a container's key column	12.4.21
`value_options`	database options for a container's value column	12.4.22
`id_column`	column name for a container's object id	12.4.23
`index_column`	column name for a container's index	12.4.24
`key_column`	column name for a container's key	12.4.25
`value_column`	column name for a container's value	12.4.26

Many of the member specifiers have corresponding value type specifiers with the same names (Section 12.3, "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.

12.4.1 `id`

The `id` specifier specifies that a 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 `id` specifier (Section 12.1.6, "`id`").

Note also that the `id` specifier cannot be used for data members of composite value types or views.

12.4.2 `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 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 `unsigned long long` as the identifier type is a safe choice.

For additional information on the automatic identifier assignment, refer to Section 3.7, "Making Objects Persistent".

Note also that the `auto` specifier cannot be specified for data members of composite value types or views.

12.4.3 `type`

The `type` specifier specifies the native database type that should be used for a data member. For example:

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

  #pragma db type("INT")
  bool married_;
};
```

The `null` and `not_null` (Section 12.4.4, "`null`/`not_null`") specifiers can be used to control the NULL semantics of a data member.

12.4.4 `null`/`not_null`

The `null` and `not_null` specifiers specify that a 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 12.4.1, "`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:

```
using std::tr1::shared_ptr;

#pragma db object
class person
{
  ...

  #pragma db null
  std::string name_;
};

#pragma db object
class account
{
  ...

  #pragma db not_null
  shared_ptr<person> holder_;
};
```

The `NULL` semantics can also be specified on the per-type basis (Section 12.3.3, "`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".

12.4.5 `default`

The `default` specifier specifies the database default value that should be used for a 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 12.4.6, "`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 to suitable database types as performed by the ODB compiler. If you have mapped a C++ enum 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 12.3.4, "`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 12.4.1, "`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.

12.4.6 `options`

The `options` specifier specifies additional column definition options that should be used for a data member. For example:

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

  #pragma db options("UNIQUE")
  std::string email_; // Mapped to TEXT NOT NULL UNIQUE.
};
```

Options can also be specified on the per-type basis (Section 12.3.5, "`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("UNIQUE")
  std::string last_;  // TEXT NOT NULL COLLATE binary UNIQUE

  #pragma db options()
  std::string title_; // TEXT NOT NULL

  #pragma db options() options("UNIQUE")
  std::string email_; // TEXT NOT NULL UNIQUE
};
```

ODB provides dedicated specifiers for specifying column types (Section 12.4.3, "`type`"), `NULL` constraints (Section 12.4.4, "`null`/`not_null`"), and default values (Section 12.4.5, "`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.

12.4.7 `column` (object, composite value)

The `column` specifier specifies the column name that should be used to store a 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 id_;
};
```

For a member of a composite value type, the `column` specifier specifies the column name prefix. Refer to Section 7.2.1, "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.

12.4.8 `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 a data member of a view class. For more information, refer to Section 9.1, "Object Views" and Section 9.2, "Table Views".

12.4.9 `transient`

The `transient` specifier instructs the ODB compiler not to store a 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.

12.4.10 `readonly`

The `readonly` specifier specifies that a 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 3.9, "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 12.4.1, "`id`") and inverse (Section 12.4.11, "`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::auto_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 12.3.6, "`readonly`") as well as whole objects (Section 12.1.4, "`readonly`") as read-only.

12.4.11 `inverse`

The `inverse` specifier specifies that a 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:

```
using std::tr1::shared_ptr;
using std::tr1::weak_ptr;

class person;

#pragma db object pointer(shared_ptr)
class employer
{
  ...

  std::vector<shared_ptr<person> > employees_;
};

#pragma db object pointer(shared_ptr)
class person
{
  ...
```

```
  #pragma db inverse(employee_)
  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 12.4.13, "`unordered`").

For a more detailed discussion of inverse members, refer to Section 6.2, "Bidirectional Relationships".

12.4.12 `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 12.1.5, "`optimistic`"). For example:

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

  #pragma db version
  unsigned 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 `unsigned long long` as the version type is a safe choice.

For a more detailed discussion of optimistic concurrency, refer to Chapter 11, "Optimistic Concurrency".

12.4.13 `unordered`

The `unordered` specifier specifies that a 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".

12.4.14 `table`

The `table` specifier specifies the table name that should be used to store the contents of a 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 7.2.1, "Composite Value Column and Table Names" for details.

12.4.15 `index_type`

The `index_type` specifier specifies the native database type that should be used for an ordered container's index column of a data member. The semantics of `index_type` are similar to those of the `type` specifier (Section 12.4.3, "`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_;
};
```

12.4.16 key_type

The key_type specifier specifies the native database type that should be used for a map container's key column of a data member. The semantics of key_type are similar to those of the type specifier (Section 12.4.3, "type"). For example:

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

  #pragma db key_type("INT UNSIGNED")
  std::map<unsigned short, float> age_weight_map_;
};
```

12.4.17 value_type

The value_type specifier specifies the native database type that should be used for a container's value column of a data member. The semantics of value_type are similar to those of the type specifier (Section 12.4.3, "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 12.4.18, "value_null/value_not_null") specifiers can be used to control the NULL semantics of a value column.

12.4.18 `value_null`/`value_not_null`

The `value_null` and `value_not_null` specifiers specify that a container's element value for a 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 12.4.4, "`null`/`not_null`"). For example:

```
using std::tr1::shared_ptr;

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

#pragma db object
class account
{
  ...

  #pragma db value_not_null
  std::vector<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.

12.4.19 `id_options`

The `id_options` specifier specifies additional column definition options that should be used for a container's id column of a 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 12.4.6, "`options`").

12.4.20 `index_options`

The `index_options` specifier specifies additional column definition options that should be used for a container's index column of a 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 12.4.6, "`options`").

12.4.21 `key_options`

The `key_options` specifier specifies additional column definition options that should be used for a container's key column of a 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 12.4.6, "`options`").

12.4.22 `value_options`

The `value_options` specifier specifies additional column definition options that should be used for a container's value column of a 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 12.4.6, "`options`").

12.4.23 `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 a data member. The semantics of `id_column` are similar to those of the `column` specifier (Section 12.4.7, "`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.

12.4.24 `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 a data member. The semantics of `index_column` are similar to those of the `column` specifier (Section 12.4.7, "`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.

12.4.25 `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 a data member. The semantics of `key_column` are similar to those of the `column` specifier (Section 12.4.7, "`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.

12.4.26 `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 a data member. The semantics of `value_column` are similar to those of the `column` specifier (Section 12.4.7, "`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.

12.5 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 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 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.

12.5.1 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 ...
```

12.5.2 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 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 id_;
};

#pragma warning (pop)
```

12.5.3 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 ...
```

12.5.4 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 ...
```

12.5.5 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 ...
```

PART II DATABASE SYSTEMS

Part II covers topics specific to the database system implementations and their support in ODB. In particular, it describes 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.

13 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.

13.1 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 12, "ODB Pragma Language").

C++ Type	MySQL Type	Default `NULL` Semantics
`bool`	`TINYINT(1)`	`NOT NULL`
`char`	`TINYINT`	`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(255)`	`NOT NULL`

Note that the `std::string` type is mapped differently depending on whether the 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(255)` MySQL type. Otherwise, it is mapped to `TEXT`.

The MySQL ODB runtime library also provides support for mapping the `std::string` type to the MySQL `CHAR`, `NCHAR`, and `NVARCHAR` types, as well as for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, and `unsigned char[N]` types to the MySQL BLOB types. However, these mappings are not enabled by default (in particular, by default, `std::vector` 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 12.4.3, "`type`"), for example:

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

  #pragma db type("CHAR(2)")
  std::string state_;

  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("BLOB")
  unsigned char[16] uuid_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
typedef std::vector<char> buffer;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Additionally, by default, C++ enumerations are automatically mapped to a suitable MySQL type. Contiguous enumerations with the zero first enumerator are mapped to the MySQL `ENUM` type. All other enumerations are mapped to `INT` or `INT UNSIGNED`. In both cases the default `NULL` semantics is `NOT NULL`. For example:

```
enum color {red, green, blue};
enum taste
{
  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 INT UNSIGNED NOT NULL.
};
```

13.2 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_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_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_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_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_ptr<connection_factory> = 0);

  database (int& argc,
            char* argv[],
            bool erase = false,
            const std::string& charset = "",
            unsigned long client_flags = 0,
            std::auto_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 13.4, "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. 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 13.3, "MySQL Connection and Connection Factory".

13.3 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 ();
    };

    typedef details::shared_ptr<connection> connection_ptr;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.5, "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 connection.

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*);
      };

      typedef details::shared_ptr<pooled_connection> pooled_connection_ptr;

      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[])
{
  auto_ptr<odb::mysql::connection_factory> f (
    new odb::mysql::connection_pool_factory (20));
```

```
  auto_ptr<odb::database> db (
    new mysql::database (argc, argv, false, 0, f));
}
```

13.4 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.

13.5 MySQL Limitations

The following sections describe MySQL-specific limitations imposed by the current MySQL and ODB runtime versions.

13.5.1 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, schemas generated by the ODB compiler for MySQL have foreign key definitions commented out. They are retained only for documentation.

14 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.

14.1 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 12, "ODB Pragma Language").

C++ Type	SQLite Type	Default `NULL` Semantics
`bool`	`INTEGER`	`NOT NULL`
`char`	`INTEGER`	`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`	`NOT NULL`
`double`	`REAL`	`NOT NULL`
`std::string`	`TEXT`	`NOT NULL`

The SQLite ODB runtime library also provides support for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, and `unsigned char[N]` types to the SQLite BLOB type. However, this mapping is not enabled by default (in particular, by default, `std::vector` will be treated as a container). To enable the BLOB mapping for these types we need to specify the database type explicitly using the `db type` pragma (Section 12.4.3, "`type`"), for example:

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

  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("BLOB")
  unsigned char[16] uuid_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
typedef std::vector<char> buffer;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Additionally, by default, C++ enumerations are automatically mapped to the SQLite `INTEGER` type with the default `NULL` semantics being `NOT NULL`.

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 long` values will be represented in the database as negative values.

14.2 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,
                std::auto_ptr<connection_factory> = 0);

      database (int& argc,
                char* argv[],
                bool erase = false,
                int flags = SQLITE_OPEN_READWRITE,
                bool foreign_keys = true,
                std::auto_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. 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 14.5.3, "Foreign Key Constraints" for more information on foreign keys.

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:

```
auto_ptr<odb::database> db (
  new odb::sqlite::database (
    "test.db",
    SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE));
```

The second 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 second constructor throws the `odb::sqlite::cli_exception` exception if the SQLite option values are missing or invalid. See Section 14.4, "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 second constructor.

The last argument to both constructors is a pointer to the connection factory. 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 3.4, "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 14.3, "SQLite Connection and Connection Factory".

14.3 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 ();
    };

    typedef details::shared_ptr<connection> connection_ptr;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.5, "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 14.2, "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 three implementations of the `connection_factory` interface provided by the SQLite ODB runtime library are `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 `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_type&);
        single_connection (database_type&, MYSQL*);
      };

      typedef details::shared_ptr<single_connection> single_connection_ptr;

      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*);
      };

      typedef details::shared_ptr<pooled_connection> pooled_connection_ptr;

      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[])
{
  auto_ptr<odb::sqlite::connection_factory> f (
    new odb::sqlite::connection_pool_factory (20));

  auto_ptr<odb::database> db (
    new sqlite::database (argc, argv, false, SQLITE_OPEN_READWRITE, f));
}
```

14.4 SQLite Exceptions

The SQLite ODB runtime library defines the following SQLite-specific exceptions:

```
namespace odb
{
  namespace sqlite
  {
    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::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.

14.5 SQLite Limitations

The following sections describe SQLite-specific limitations imposed by the current SQLite and ODB runtime versions.

14.5.1 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 3.13, "ODB Exceptions") is always thrown. Future versions of the SQLite ODB runtime library may add support for result caching.

14.5.2 Automatic Assignment of Object Ids

Due to SQLite API limitations, every automatically assigned object id (Section 12.4.2, "`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 12.4.3, "`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_;
};
```

14.5.3 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 3.10, "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 relies on 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 must declare such constraints as `DEFERRABLE INITIALLY DEFERRED`, as shown in the following example. Schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer INTEGER REFERENCES Employer(id)
           DEFERRABLE INITIALLY DEFERRED);
```

14.5.4 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_not_persistent` exception (Section 3.13, "ODB Exceptions").

14.5.5 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.

15 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.

15.1 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 12, "ODB Pragma Language").

C++ Type	PostgreSQL Type	Default `NULL` Semantics
`bool`	`BOOLEAN`	`NOT NULL`
`char`	`SMALLINT`	`NOT NULL`
`signed char`	`SMALLINT`	`NOT NULL`
`unsigned char`	`SMALLINT`	`NOT NULL`
`short`	`SMALLINT NULL`	`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`

The PostgreSQL ODB runtime library also provides support for mapping the `std::string` type to the PostgreSQL `CHAR` and `VARCHAR` types, as well as for mapping the `std::vector<char>`, `std::vector<unsigned char>`, `char[N]`, and `unsigned char[N]` types to the PostgreSQL `BYTEA` type. However, these mappings are not enabled by default (in particular, by default, `std::vector` 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 12.4.3, "`type`"), for example:

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

  #pragma db type("CHAR(2)")
  std::string state_;

  #pragma db type("BYTEA")
  std::vector<char> buf_;

  #pragma db type("BYTEA")
  unsigned char[16] uuid_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
typedef std::vector<char> buffer;
#pragma db value(buffer) type("BYTEA")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BYTEA.
};
```

Additionally, by default, C++ enumerations are automatically mapped to `INTEGER` with the default `NULL` semantics being `NOT NULL`.

Note also that because PostgreSQL does not support unsigned integers, the `unsigned short`, `unsigned int`, and `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.

15.2 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_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_ptr<connection_factory> = 0);

      database (const std::string& conninfo,
                std::auto_ptr<connection_factory> = 0);

      database (int& argc,
                char* argv[],
                bool erase = false,
                const std::string& extra_conninfo = "",
                std::auto_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 15.4, "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. 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 15.3, "PostgreSQL Connection and Connection Factory".

15.3 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 ();
    };

    typedef details::shared_ptr<connection> connection_ptr;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.5, "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*);
      };

      typedef details::shared_ptr<pooled_connection> pooled_connection_ptr;

      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[])
{
  auto_ptr<odb::pgsql::connection_factory> f (
    new odb::pgsql::connection_pool_factory (20));

  auto_ptr<odb::database> db (
    new pgsql::database (argc, argv, false, "", f));
}
```

15.4 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.

15.5 PostgreSQL Limitations

The following sections describe PostgreSQL-specific limitations imposed by the current PostgreSQL and ODB runtime versions.

15.5.1 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.

15.5.2 Foreign Key Constraints

ODB relies on 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 must declare such constraints as `INITIALLY DEFERRED`, as shown in the following example. Schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer BIGINT REFERENCES Employer(id) INITIALLY DEFERRED);
```

15.5.3 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_not_persistent` exception (Section 3.13, "ODB Exceptions").

15.5.4 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
```

15.5.5 Timezones

ODB does not currently support the PostgreSQL date-time types with timezone information.

15.5.6 `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.

16 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.

16.1 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 12, "ODB Pragma Language").

C++ Type	Oracle Type	Default `NULL` Semantics
`bool`	`NUMBER(1)`	`NOT NULL`
`char`	`NUMBER(3)`	`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(4000)`	`NULL`

In Oracle empty VARCHAR2 and NVARCHAR2 strings are represented as NULL values. As a result, in the generated schema, columns of these types are always declared as NULL, even if explicitly declared as NOT NULL with the db not_null pragma (Section 12.4.4, "null/not_null").

The Oracle ODB runtime library also provides support for mapping the std::string type to the Oracle CHAR, NCHAR, NVARCHAR2, CLOB and NCLOB types, as well as for mapping the std::vector<char>, std::vector<unsigned char>, char[N], and 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 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 12.4.3, "type"), for example:

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

  #pragma db type ("CLOB")
  std::string str_;

  #pragma db type("BLOB")
  std::vector<char> buf_;

  #pragma db type("RAW(16)")
  unsigned char[16] uuid_;
};
```

Alternatively, this can be done on the per-type basis, for example:

```
typedef std::vector<char> buffer;
#pragma db value(buffer) type("BLOB")

#pragma db object
class object
{
  ...

  buffer buf_; // Mapped to BLOB.
};
```

Additionally, by default, C++ enumerations are automatically mapped to NUMBER(10) with the default NULL semantics being NOT NULL.

16.2 Oracle Database Class

The Oracle `database` class 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_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_ptr<connection_factory> = 0);

      database (int& argc,
                char* argv[],
                bool erase = false,
                ub2 charset = 0,
                ub2 ncharset = 0,
                OCIEnv* environment = 0,
                std::auto_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 () const;

    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 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.

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 16.4, "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 a connection factory. 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 16.3, "Oracle Connection and Connection Factory".

16.3 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 ();
    };

    typedef details::shared_ptr<connection> connection_ptr;
  }
}
```

For more information on the `odb::connection` interface, refer to Section 3.5, "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.

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*);
      };

      typedef details::shared_ptr<pooled_connection> pooled_connection_ptr;

      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[])
{
  auto_ptr<odb::oracle::connection_factory> f (
    new odb::oracle::connection_pool_factory (20));

  auto_ptr<odb::database> db (
    new oracle::database (argc, argv, false, 0, 0, 0, f));
}
```

16.4 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;
      };

      typedef std::vector<record> records;

      typedef records::size_type size_type;
      typedef records::const_iterator 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.

16.5 Oracle Limitations

The following sections describe Oracle-specific limitations imposed by the current Oracle and ODB runtime versions.

16.5.1 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 12, "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_;
};
```

16.5.2 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 3.13, "ODB Exceptions") is always thrown. Future versions of the Oracle ODB runtime library may add support for result caching.

16.5.3 Foreign Key Constraints

ODB relies on 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 must declare such constraints as `INITIALLY DEFERRED`, as shown in the following example. Schemas generated by the ODB compiler meet this requirement automatically.

```
CREATE TABLE Employee (
  ...
  employer NUMBER(20) REFERENCES Employer(id)
           DEFERRABLE INITIALLY DEFERRED);
```

16.5.4 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_not_persistent` exception (Section 3.13, "ODB Exceptions").

16.5.5 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.

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.

16.5.6 Timezones

ODB does not currently support the Oracle date-time types with timezone information.

16.5.7 `LONG` Types

ODB does not support the deprecated Oracle `LONG` and `LONG RAW` data types.

PART III PROFILES

Part III covers the integration of ODB with popular C++ frameworks and libraries. It consists of the following chapters.

17 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.

18 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 3.13, "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`. The concrete exceptions that can be thrown by the Boost sub-profiles are described in the following sections.

18.1 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 3.2, "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 the description of the lazy pointer interface and semantics refer

to Section 6.3, "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 3.2, "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.

18.2 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_;
};
```

18.3 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.

18.4 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_;
};
```

The 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.

18.4.1 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 12.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class person
{
  ...
  #pragma db type("TIMESTAMP") not_null
  boost::posix_time::ptime updated_;
};
```

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.

18.4.2 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 12.4.3, "`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.

18.4.3 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.

18.4.4 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.

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.

19 Qt Profile

The ODB profile implementation for Qt is provided by the `libodb-qt` library and 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 3.13, "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`. The concrete exceptions that can be thrown by the Qt sub-profiles are described in the following sections.

19.1 Basic Types

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` and `QByteArray`. 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_;
};
```

19.1.1 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(255)`	`NULL`
`QByteArray`	`BLOB`	`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 the 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(255)` 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 12.4.3, "type"), as shown in the following example:

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

  #pragma db type("CHAR(2)") not_null
  QString licenseState_;
};
```

19.1.2 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`

Instances of the `QString` and `QByteArray` types are stored as a NULL value if their `isNull()` member function returns `true`.

19.1.3 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`

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 12.4.3, "type"), as shown in the following example:

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

  #pragma db type("CHAR(2)") not_null
  QString licenseState_;
};
```

19.1.4 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(4000)`	`NULL`
`QByteArray`	`BLOB`	`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 12.4.3, "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_;
};
```

19.2 Smart Pointers

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 3.2, "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 the description of the lazy pointer interface and semantics refer to Section 6.3, "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 3.2, "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.

19.3 Containers Library

The `container` 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_;
};
```

19.4 Date Time Types

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.

19.4.1 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 12.4.3, "`type`"), as shown in the following example:

```
#pragma db object
class Person
{
  ...
  #pragma db type("TIMESTAMP") not_null
  QDateTime updated_;
};
```

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.

19.4.2 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 12.4.3, "`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.

19.4.3 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.

19.4.4 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.