C++ Object Persistence with ODB

Revision 1.0, September 2010

This revision of the manual describes ODB 1.0.0 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.

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.

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.

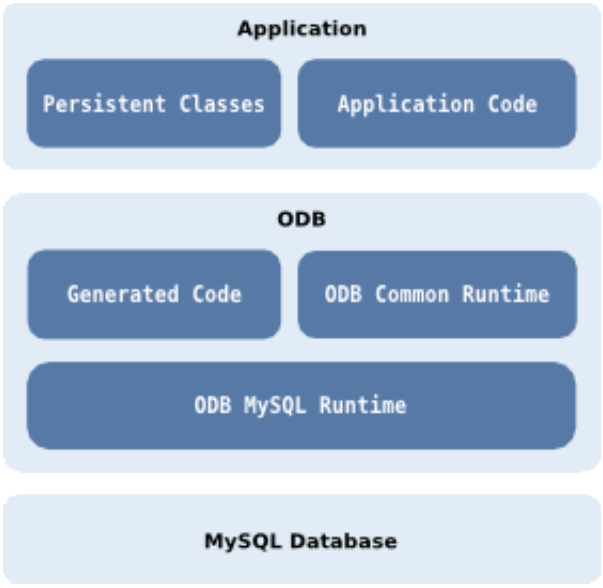
1.1 Architecture and Workflow

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.

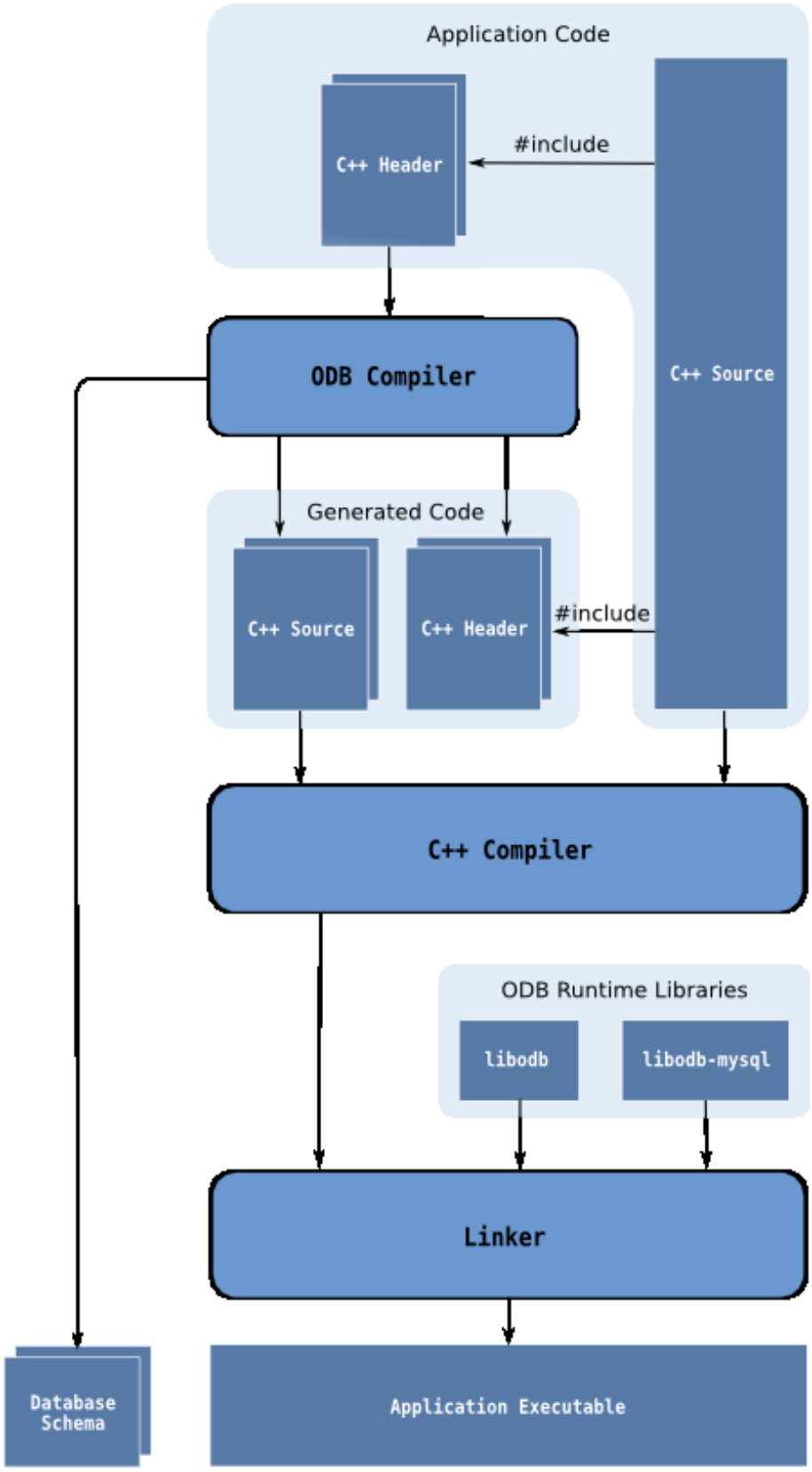
The common runtime library defines database system-independent interfaces that your application can use to manipulate persistent objects. The database-specific runtime library provides implementations of these interfaces for a concrete database as well as other database-specific utilities that are used 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 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`. Note that after executing this command, all the data stored in the `odb_test` database will be deleted.

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;

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

      db->persist (john);
      db->persist (jane);
      db->persist (joe);

      t.commit ();

      // Save object ids for later use.
      //
      john_id = john.id ();
      jane_id = jane.id ();
      joe_id = joe.id ();
    }
  }
  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.

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 (see Section 6.1.2, "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.

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.

After the transaction has been committed, we save the objects' identifiers in local variables. We will use them later in this chapter to perform other database operations on our persistent objects. You might have noticed that our `person` class doesn't have the `id()` function that we use here. To make our code compile we need to add a simple accessor with this name that returns the value of the `id_` data member.

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 (see Section 3.8, "ODB Exceptions") and printing the diagnostics.

Let's now compile (see 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
```

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.

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 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 ();
}
```

2.8 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 continue with the discussion of the `odb::database` class in Section 3.2

and transactions in Section 3.3. 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 or values. An object has an identifier, called *object id*, that is unique among all instances of an object type within a database. An object consists of data members which are either values or references to other objects. 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 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*. Otherwise, the value is called *simple value* and its type — a *simple value type*. 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.

Seeing 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.

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
{

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

These two pragmas are the minimum required to declare a persistent class. Other pragmas can be used to fine-tune the database-related properties of a class and its members (see Chapter 5, "ODB Pragma Language").

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 you will need to provide the mapping to the database system type and, possibly, the code to convert between the two. For more information on this refer to Section 5.3, "Value Type Pragmas". Composite value types are not yet supported by ODB and we will not discuss them further in this revision of the manual.

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 references 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 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. You 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. You 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 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 Chapter 6, "Database Systems".

3.3 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 the `database::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. The `odb::transaction` class has the following interface:

```
namespace odb
{
  class transaction
  {
  public:
    typedef odb::database database_type;

    void
    commit ();

    void
    rollback ();

    database_type&
    database ();

    static transaction&
    current ();

    static bool
    has_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 you try to commit or roll back a finalized transaction, the `odb::transaction_already_finalized` exception is thrown.

The `database()` function returns the database this transaction is working on. The `current()` static function returns the currently active transaction for this thread. If there is no active transaction, this function throws the `odb::not_in_transaction` exception. You can

check whether there is a transaction in effect in this thread using the `has_current()` static function.

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` exception being thrown from one of the database functions. You would normally handle a deadlock by restarting the transaction, for example:

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

    ...

    t.commit ();
    break;
  }
  catch (const odb::deadlock&)
  {
    continue;
  }
}
```

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;
  }
}
```

3.4 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 two 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 (T& object);
```

The first `persist()` function expects a constant reference to an instance being persisted and is used on objects with application-assigned object ids (see Section 5.4, "Data Member Pragmas"). The second function expects an unrestricted reference and, if the object id is assigned by the database, it updates the id member of the passed instance with the assigned value. Both functions return the object id of the newly persistent 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 `odb::object_traits` template used in the signature above is part of the database support code generated by the ODB compiler.

The following example shows how we can call these functions:

```
person john ("John", "Doe", 33);
person jane ("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 made available to other threads in your application and to other applications as soon as possible.

3.5 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 ();
```

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 (see Chapter 4, "Querying the Database"). Note, however, that loading an object’s state using its identifier can be significantly faster than executing a query.

3.6 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:

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

If the object passed to this function does not exist in the database, `update()` throws the `odb::object_not_persistent` exception.

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 ();
}
```

3.7 Deleting Persistent Objects

To delete a persistent object's state from the database we use the `database::erase()` function template. 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 typename object_traits<T>::id_type& id);
```

The first `erase()` function uses an object itself to delete its state from the database. Note that the passed object is unchanged. It simply becomes transient. The second function uses the object id to identify the object to be deleted. If the object does not exist in the database, both functions throw the `odb::object_not_persistent` exception.

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

```
const person& john = ...

transaction t (db->begin ());

db->erase (john);
db->erase<person> (jane_id);

t.commit ();
```

3.8 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 inherits 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 you 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 already_in_transaction: odb::exception
  {
    virtual const char*
    what () const throw ();
  };

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

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

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

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

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

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

  struct database_exception: odb::exception
  {
  };
}
```

The first four exceptions (`already_in_transaction`, `not_in_transaction`, `transaction_already_finalized`, and `deadlock`) are thrown by the `odb::transaction` class and are discussed in Section 3.3, "Transactions".

The `object_already_persistent` exception is thrown by the `persist()` database function. See Section 3.4, "Making Objects Persistent" for details.

The `object_not_persistent` exception is thrown by the `load()` and `update()` database functions. Refer to Section 3.5, "Loading Persistent Objects" and Section 3.6, "Updating Persistent Objects" for more information.

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

The `database_exception` is a base class for all database system-specific exceptions that are thrown by the database system-specific runtime library. See Chapter 6, "Database Systems" for more information.

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 you don't know the identifiers of the objects that you are looking for, you can use queries to search the database for objects matching certain criteria. The ODB query facility is optional and you 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 you 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 you 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 you 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.

You 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 you 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. You 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 you 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 you 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 you would create a named query instance if you are planning to run the same query multiple times and would use the in-line version for those that are executed only once.

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 you 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 you 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 you have a large number of objects in the result or if you 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 you 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 you have two iterators pointing to the current position and then you 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, you 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);
  };
}
```

When you 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 you 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 you 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 you 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;
}
```

5 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`, `value`, and `member`. Pragmas with the `object` qualifier describe persistent object types. It tells the ODB compiler that the C++ class it describes is a persistent class. Similarly, pragmas with the `value` qualifier describes value types and the `member` qualifier is used to describe data members of persistent object 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, you 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 NOT NULL")

#pragma db object
class person
{
  ...
private:
  bool married_; // Mapped to INT NOT NULL database type.
  ...
};
```

5.1 C++ Compiler Warnings

The C++ header file that defines your 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 your application. Some C++ compilers issue warnings about pragmas that they do not 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 you 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
{
  ...
private:
  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
{
  ...
private:
  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.

5.1.1 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, you can add the `-Wno-unknown-pragmas` option after `-Wall`, for example:

```
g++ -Wall -Wno-unknown-pragmas ...
```

5.1.2 Visual C++

Microsoft Visual C++ issues an unknown pragma warning (C4068) at warning level 1 or higher. This means that unless you have disabled the warnings altogether (level 0), you will see this warning.

To disable this warning via the compiler command line, you 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 you can enter warning numbers that should be disabled. Simply enter 4068 into this field.

You 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
{
  ...
private:
  #pragma db id
  unsigned long id_;
  ...
};

#pragma warning (pop)
```

5.1.3 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 you can add the `-erroff=unknownpragma` option anywhere on the command line, for example:

```
CC +w -erroff=unknownpragma ...
```

5.1.4 IBM XL C++

IBM XL C++ issues an unknown pragma warning (1540-1401) by default. To disable this warning you can add the `-qsuppress=1540-1401` command line option, for example:

```
xlC -qsuppress=1540-1401 ...
```

5.2 Object Type Pragmas

A pragma with the `object` qualifier declares a C++ class as a persistent object type. The qualifier can be optionally followed by the `table` specifier.

5.2.1 `table`

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

5.3 Value Type Pragmas

A pragma with the `value` qualifier describes a value type and can be optionally followed by the `type` specifier.

5.3.1 `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 NOT NULL")

#pragma db object
class person
{
  ...
private:
  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 Chapter 6, "Database Systems".

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) NOT NULL")
```

The possible database values for the C++ `true` value could be `"true"`, or `"TRUE"`, or `"True"`. Or, maybe, all of the above are 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.

It is also possible to change the database type mapping for individual members, as described in Section 5.4, "Data Member Pragmas".

5.4 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`	the member is an object id	5.4.1
`auto`	id is assigned by the database	5.4.2
`type`	the database type for the member	5.4.3
`column`	the column name for the member	5.4.4
`transient`	the member is not stored in the database	5.4.5

5.4.1 `id`

The `id` specifier specifies that the data member contains the object id. Every persistent class must have a member designated as an object’s identifier. For example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id
  std::string email_;
  ...
};
```

In a relational database, an identifier member is mapped to a primary key.

5.4.2 `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
{
  ...
private:
  #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 3.4, "Making Objects Persistent".

5.4.3 `type`

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

```
#pragma db object
class person
{
  ...
private:
  #pragma db type("INT NOT NULL")
  bool married_;
  ...
};
```

The behavior of this specifier for members is similar to that for value types. The only difference is the scope. The value type pragma applies to all members with this value type that don't have their own `type` specifiers, while the member pragma applies only to a single member. For more information on the semantics of this specifier, refer to Section 5.3, "Value Type Pragmas".

5.4.4 `column`

The `column` specifier specifies the column name that should be used to store this member in a relational database. For example:

```
#pragma db object
class person
{
  ...
private:
  #pragma db id column("person_id")
  unsigned long id_;
  ...
};
```

If the column name is not specified, it is derived from the member name by removing the common member name decorations, such as leading and trailing underscores, the `m_` prefix, etc.

5.4.5 `transient`

The `transient` specifier instructs the ODB compiler not to store the data member in the database. For example:

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

6 Database Systems

This chapter 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.

6.1 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.

6.1.1 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 Pragmas Language (see Chapter 5, "ODB Pragma Language").

C++ Type	MySQL type
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 NOT NULL/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 `VARCHAR(255) NOT NULL` MySQL type. Otherwise, it is mapped to `TEXT NOT NULL`.

6.1.2 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,
          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,
          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,
          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,
          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,
          unsigned long client_flags = 0,
          std::auto_ptr<connection_factory> = 0);

database (int& argc,
          char* argv[],
          bool erase = false,
          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;

      unsigned long
      client_flags () const;

    public:
      details::shared_ptr<mysql::connection>
      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 you 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 `client_flags` argument allows you 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 you to specify some or all of the database options in a file with each option appearing on a separate line followed by 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 6.1.4, "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 the pointer to the connection factory. If you 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 allow you to query the parameters of the `database` instance.

The `connection()` function returns the MySQL database connection encapsulated by the `odb::mysql::connection` class. Normally, you wouldn't call this function directly and instead let the ODB runtime manage the database connections. However, if for some reason you need to access the underlying MySQL connection handle, refer to the MySQL ODB runtime source code for the interface of the `connection` class.

6.1.3 Connection Factory

The `connection_factory` abstract class has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class connection_factory
    {
```

```
    public:
      virtual void
      database (mysql::database&) = 0;

      virtual details::shared_ptr<mysql::connection>
      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 the `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 `connection_pool_factory` class implements a connection pool. It has the following interface:

```
namespace odb
{
  namespace mysql
  {
    class connection_pool_factory: public connection_factory
    {
      connection_pool_factory (std::size_t max_connections = 0,
                               std::size_t min_connections = 0)
    };
};
```

The `max_connections` argument 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 calling thread is blocked until a connection becomes available.

When a connection is released, the pool factory first checks if there are blocked threads waiting for a connection. If so, 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 the `min_connections` number and there are no threads 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.

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/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));
}
```

6.1.4 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.