EyeDB C++ Binding

Version 2.8.0

January 2006

Published by SYSRA
30, avenue Général Leclerc
91330 Yerres - France

home page: http://www.eyedb.org

Contents

The EyeDB C++ Binding

The C++ binding maps the EyeDB object model into C++ by introducing a generic API and a tool to generate a speficic C++ API from a given schema, built upon the generic API.

The generic C++ API is made up of about one hundred of classes such as some abstract classes as the `object` and `class` classes and some more concrete classes such as the `database` and `image` classes.

Each type in the EyeDB object model is implemented as a C++ class whithin the C++ API: there is a one for one mapping between the object model and the C++ API.
This mapping follows a very simple naming scheme: each C++ class mapped from a type has the name of this type prefixed by the namespace `eyedb`.
For instance, the `object` type in the EyeDB object model is mapped to the `eyedb::Object` C++ class and the `agregat` type is mapped to the `eyedb::Agregat` C++ class.

To avoid writing each time the full qualified type name (i.e. `eyedb::`*type*), you may use the C++ instruction `using namespace eyedb`.

We are going to introduce the main classes and methods through some simple examples.

1 The Generic C++ API

1.1 Initialization

The minimal EyeDB C++ program is as follows:

```
#include <eyedb/eyedb.h>

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);
  // ...
  eyedb::release();
  return 0;
}
```

A few remarks about this code:

1. the file `eyedb/eyedb.h` contains the whole EyeDB C++ API; except for some specific administration or hacker tasks, it is not necessary to include any other eyedb files.

2. the EyeDB C++ layer must be initialized using one of the static method `init` of the namespace `eyedb`:

   (a) static void eyedb::init()

   (b) static void eyedb::init(int &argc, char *argv[])

   The first method only initializes the EyeDB layer while the second one performs also some command line option processing. For instance, the option `--port=<port>` allows you to use the port `<port>` as the default connection port to the EyeDB server, while the option `-logdev=stderr` displays log information on the standard error.

   The option `--help-eyedb-options` displays a brief usage for each standard options:

   ```
   -U <user>|@, --user=<user>|@       User name
   ```

```
-P [<passwd>], --passwd[=<passwd>] Password
--host=<host>                      eyedbd host
--port=<port>                      eyedbd port
--inet                             Use the tcp_port variable if port is not set
--dbm=<dbmfile>                    EYEDBDBM database file
--conf=<conffile>                  Configuration file
--logdev=<logfile>                 Output log file
--logmask=<mask>                   Output log mask
--logdate=on|off                   Control date display in output log
--logtimer=on|off                  Control timer display in output log
--logpid=on|off                    Control pid display in output log
--logprog=on|off                   Control progname display in output log
--error-policy=<value>             Control error policy: status|exception|abort|stop|echo
--trans-def-mag=<magorder>         Default transaction magnitude order
--arch                             Display the client architecture
-v, --version                      Display the version
--help-eyedb-options               Display this message
```

Note that all the standard command line options recognized in the `argc/argv` array are suppressed from this array by `eyedb::init(int &argc, char *argv[])`.

3. the last statement `eyedb::release()` allows you to release all the EyeDB allocated resources and to close opened databases and connections. Note that this statement is optionnal as all EyeDB allocated resources, opened databases and connections will be automatically released or closed in the `exit()` function.

1.2 Connection Setting-up

To manage objects within a database we need to open this database. But before opening any database we need to establish a connection with the EyeDB server.
The connection to the EyeDB server is realized through the `eyedb::Connection` class as follows:

```
eyedb::Connection conn;
conn.open();
```

A few remarks about this code:

1. the construction of an `eyedb::Connection` instance (first line of code) does not perform any actual actions: it only constructs a runtime instance.

2. to establish the connection, one needs to use the `eyedb::Connection::open(const char *host=0, const char *port=0)` method. This method has two optionnal arguments: `host` and `port`.
   If these arguments are not specified, their values are taken from the configuration or from the command line options `--host=<host>` and `--port=<port>` if specified.

3. in case of an error happened during the connection setting-up, a status is returned or an exception is raised depending on the chosen error policy. The default error policy is the `status error policy` which means that each EyeDB method returns a status implemented by the `eyedb::Status` class. The special status `eyedb::Success` (in fact a null pointer) means that the operation has been performed successfully:

   - ```
     eyedb::Status s;
     eyedb::Connection conn;
     s = conn.open();
     if (s) {
       cerr << status;
       return 1;
     }
     ```

   The `exception error policy` means that each EyeDB method throws an exception, implemented by the class `eyedb::Exception`, when an error happened:

   - ```
     try {
       eyedb::Connection conn;
       conn.open();
     }
     catch(eyedb::Exception &e) {
       cerr << e;
       return 1;
     }
     ```

Note that `eyedb::Status` is an alias for `const eyedb::Exception *`. To use the `exception error policy`, one needs to call the following method before any operation:

```
eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);
```

Although the exception error policy is not currently the default one in EyeDB, we recommend to use it: it makes code clearer and safer.
In the following examples we use the `exception error policy` to avoid any error management noise in the introduced C++ code.

1.3 Database Opening

To open a database one uses the `eyedb::Database` class as follows:

```
const char *dbname = argv[1];
eyedb::Database db(dbname);
db.open(&conn, eyedb::Database::DBRW);
```

1. as the `eyedb::Connection` constructor, the `eyedb::Database` constructor does not perform any actual operation: it constructs a runtime instance.
2. to open a database one uses the `eyedb::Database::open` methods which takes the following arguments:
   (a) a pointer to an opened `eyedb::Connection` instance.
   (b) the opening flag which can be either `eyedb::Database::DBRead` for read-only opening or `eyedb::Database::DBRW` for read-write opening.
   Note that there are a dozen of opening modes that are introduced in the reference manual.
   (c) the user authentication
   (d) the password authentication

   The two last arguments are optionnal: if not specified, their values are taken from the configuration file or from the command line options `--user=<user>` and `--passwd=<passwd>`, or from the standard input when using `--passwd` without given value.

Note that an EyeDB client can manage several connections and several databases on each connection, for instance:

```
eyedb::Connection conn_local;
conn_local.open();
eyedb::Connection conn_remote;
conn_remote.open("arzal.zoo.com", 7620);

eyedb::Database db_1("foo");
db_1.open(&conn_local, eyedb::Database::DBRW);

eyedb::Database db_2("EYEDBDBM");
db_2.open(&conn_local, eyedb::Database::DBRead, "guest", "guest");

eyedb::Database db_3("droopy");
db_2.open(&conn_remote, eyedb::Database::DBRW, "droopy", "xyztu");
```

1.4 Transaction Management

Any object operation - storing or loading for instance - within a database must be done in the scope of a transaction.

A transaction is an unit with atomicity, coherency and integrity.

1. **Atomicity** means that the transaction modifications are either realized (commit) or not realized at all (rollback or abort).
2. **Coherency** means that a transaction starts from a coherent database state, and leaves the database in a coherent state.
3. **Integrity** means that a transaction modification is not lost, even in case of a process, operating system or hardware failure.

A transaction scope is composed of a starting point, `transactionBegin`, and an ending point, `transactionCommit` or `transactionAbort`:

```
eyedb::Database db(dbname);
db.open(&conn, eyedb::Database::DBRW);

db.transactionBegin();
// ... object operations
db.transactionCommit();
```

A call to `eyedb::Database::transactionCommit()` means that all the operations performed in the transaction scope will be stored in the database, while a call to `eyedb::Database::transactionAbort()` means that all the operations will be forgotten.

Currently, EyeDB does not support nested transactions but it allows you to write code such as:

```
db.transactionBegin();     // level 0 begin
// ... object operations
   db.transactionBegin();  // level 1 begin
   // ... object operations
   db.transactionAbort();  // level 1 abort
// ... object operations
db.transactionCommit();    // level 0 commit
```

But the abort at level 1 is without effect: it will not be performed; only the commit at level 0 will be performed.

One can give parameters to the transaction that one begins by setting an optional argument of type `eyedb::TransactionParams` to the `transactionBegin` method. The `TransactionParams` type is composed of the following public attributes :

1. the `trsmode` argument controls the transaction mode,
2. the `lockmode` argument controls the object lock policy,
3. the `recovmode` argument controls the recovery mode,
4. the `magorder` argument controls the size of the allocated tables for the transaction,
5. the `ratioalrt` argument controls the error returned if ratioalrt != 0 and trans object number ¿ ratioalrt * magorder
6. the `wait_timeout` argument controls wait timeout value.

For instance :

```
TransactionParams params; // create params with default values
params.lockmode = eyedb::ReadNWriteX; // objects are not locked for reading
                                      // and locked exclusive for writing
params.magorder = 100000000; // transaction can deal with about
                             // 100 millions of objects
db.transactionBegin(params);
```

Refer to the reference manual to get more information about these arguments.

1.5 Schema and Class Manipulation

The EyeDB C++ API provides runtime facilities to manipulate the EyeDB classes. In fact, as the class `class` inherits from the class `object`, EyeDB classes can be manipulated as objects.

A class is composed of a list of attributes, constraints, variables, methods, triggers and indexes.

The classes are gathered through a schema instance tied to each database.

A class can be a system class, for instance the class `class`, the class `object`, the class `agregat` or a user class, for instance the class `Person`, the class `Employee`.

To illustrate this object model, we are going to show how to display the user classes of a given database:

```
eyedb::Database db(dbname);
db.open(&conn, eyedb::Database::DBRW);

db.transactionBegin();
eyedb::LinkedListCursor c(db.getSchema()->getClassList());
eyedb::Class *cls;
while (c.getNext((void*&)cls))
  if (!cls->isSystem())
    cout << cls;
db.transactionCommit();
```

As shown here, this code is very simple:

1. database opening as we have seen before.
2. linked list cursor creation on the database schema class list.
3. display of each class in the list which is not a system class.

For instance, to display all the classes of type `struct` which contains an attribute named `age`:

```
eyedb::LinkedListCursor c(db.getSchema->getClassList());
eyedb::Class *cls;
while (c.getNext((void*&)cls))
  if (cls->asStructClass()) {
    int attr_cnt;
    const eyedb::Attribute **attrs = cls->getAttributes(attr_cnt);
    for (int i = 0; i < attr_cnt; i++)
      if (!strcmp(attrs[i]->getName(), "age")) {
        cout << cls;
        break;
      }
   }
```

1.6 Object Manipulation

There are two types of objects: runtime objects and database objects.
Runtime objects are the OML (Object Manipulation Language) objects, for instance C++ or Java objects, while the database objects are the objects stored in a database.

There are two types of runtime objects: persistent runtime objects and transient runtime objects.
A runtime object is persistent if it is tied to a database object. Otherwise, it is transient.

Contrary to some other OODBMS, EyeDB does not provide an automatic synchronisation between persistent runtime objects and database objects.
When setting values on a persistent runtime object, we do not modify the tied database object. We must call the `store` method on the persistent runtime object to update the tied database object.

Note that any persistent runtime object manipulation must be done in the scope of a transaction.

To illustrate object manipulations, we introduce a simple concrete example. This example will be used in the whole continuation of this chapter.

The example is as follows:

```
//
// person.odl
//

enum CivilState {
  Lady = 0x10,
  Sir  = 0x20,
  Miss = 0x40
};

class Address {
```

```
  attribute string street;
  attribute string<32> town;
  attribute string country;
};

class Person {
  attribute string name;
  attribute int age;
  attribute Address addr;
  attribute Address other_addrs[];
  attribute CivilState cstate;
  attribute Person * spouse inverse Person::spouse;
  attribute set<Car *> cars inverse owner;
  attribute array<Person *> children;

  int change_address(in string street, in string town,
                     out string oldstreet, out string oldtown);

  static int getPersonCount();
  index on name;
};

class Car {
  attribute string brand;
  attribute int num;
  Person *owner inverse cars;
};

class Employee extends Person {
  attribute long salary;
};
```

This file is located at *prefix*/`share/doc/eyedb/examples/C++Binding/schema-oriented/share/schema.odl`.

1.7 Creating Runtime Objects

Using the C++ API, we cannot create directly a database object. We must create first a runtime object.
To create a runtime object we invoke the `newObj` method of the object class.
For instance, to create a runtime `Person` object, we need to invoke the `newObj` method of the `Person` runtime class as follows:

```
  eyedb::Class *cls = db.getSchema()->getClass("Person");

  eyedb::Object *p = cls->newObj(&db);
```

The `eyedb::Class::newObj(eyedb::Database * = 0)` is the class instantiation method for both persistent and transient object.
A transient object is created using the `newObj` without any argument, while a persistent object is created using the same method with a valid database runtime pointer.

Note that as long as the `store` method has not been called, the persistent runtime object is not yet tied to a database object.
So, if we follow strictly the definition of runtime objects, it is not yet a persistent runtime object; but as soon as a runtime object is created using the `newObj` method with a valid database pointer, we will say that it is persistent.

1.8 Synchronizing Runtime Objects to Database Objects

When a persistent object is stored in the database using the `store` method, an unique object identifier `OID` is allocated to this object.
This `OID` can be acceded using the method `eyedb::Object::getOid()`, for instance to display the allocated `OID`:

```
  eyedb::Object *p = cls->newObj(&db);
  cout << "before storing: " << p->getOid() << endl;
  p->store();
  cout << "after storing: " << p->getOid() << endl;
```

The output displayed by the previous code is something as follows:

```
before storing: NULL
after storing: 1456.3.38475637:oid
```

As shown here, before the first call of the `store` method, the `OID` is not set; a `NULL` is displayed.
The created `OID` is composed of three fields:

1. the object number : `1456`
2. the database identifier : `3`
3. a magic number : `38475637`

The database identifier designates, in an unique way, a database while the object number designates, in an unique way, an object within a database.
The magic number, which is a random generated number, ensures more security in the object identification process.

1.9 Setting Attribute Values to a Runtime Object

Assume that we want to set a name and a age values to a `Person` instance. Here is a way to do so:

```
eyedb::Class *cls = db.getSchema()->getClass("Person");
eyedb::Object *p = cls->newObj(&db);

// getting attributes from class
const eyedb::Attribute *attr_name = cls->getAttribute("name");
const eyedb::Attribute *attr_age = cls->getAttribute("age");

// setting name attribute value
attr_name->setSize(p, strlen("john")+1);
attr_name->setValue(p, (eyedb::Data)"john", strlen("john")+1, 0);

// setting age attribute value
eyedb::_int32 age = 27;
attr_age->setValue(p, (eyedb::Data)&age, 1, 0);
```

We need to do a few remarks about this code:

1. to get specific named attribute within a class, we use the method `eyedb::Class::getAttribute(const char *)`. This method returns a pointer to an `eyedb::Attribute` which contains a complete description of this attribute: type, name, size, position and so on.
2. to set an attribute value for the instance `p`, we use the method `eyedb::Attribute::setValue(eyedb::Object *o, eyedb::Data data, int nb, int from)` whose arguments are as follows:
   (a) `eyedb::Object *o`: the runtime object pointer to modify.
   (b) `eyedb::Data data`: the pointer to the attribute value to set.
   (c) `int nb`: for an array, the number of values to set.
   (d) `int from`: for an array, the starting index of the values to set.
3. the `eyedb::Attribute::setSize(eyedb::Object *, eyedb::Size)` method is used for the attribute `name` because this attribute is of variable size (remember the schema description : `string name`).
   So, before setting the attribute value, we must set the size of this attribute value.
4. remember that the database object tied to this persistent object has not been changed in the database: only the transient values have been changed.
   To change the database object, one needs to use the method `eyedb::Object::store()` as follows:

   ```
   p->store();
   ```

   The `store` method allows you to synchonize the transient values of a persistent object with the database.

   To avoid all this class and attribute manipulation and to deal with direct access attribute methods, one needs to use the `eyedbodl` tool which allows you to generate specific C++ code from a specific database schema.

   For instance, using this tool, the previous code becomes:

```
Person *p = new Person(&db);

p->setName("john");
p->setAge(27);

p->store();
```

The class `Person`, the methods `setName` and `setAge` have been generated by the `eyedbodl` tool in a very simple way. Refer to the second part of this chapter `the Schema-Oriented Generated C++ API`.

1.10 Loading Database Objects

To load an object from a database, one needs to give its `OID` to the `eyedb::Database::loadObject` method, for instance:

```
eyedb::Oid oid("1456.3.38475637:oid");
eyedb::Object *o;
db.loadObject(oid, o);
cout << "object " << oid << " is of class " << o->getClass()->getName()
     << endl;
cout << o;
```

The previous code loads the object from the database, displays its oid and class name and displays the whole object.

1.11 Getting Attribute Values from a Runtime Object

The process to get attribute values from a runtime object is very similar to the process to set attribute values.
For instance to get the `name` and `age` attribute values of the previous loaded object, one can do as follows:

```
eyedb::Oid oid("1456.3.38475637:oid");
eyedb::Object *o;
db.loadObject(oid, o);

// getting attributes from class
const eyedb::Attribute *attr_name = cls->getAttribute("name");
const eyedb::Attribute *attr_age = cls->getAttribute("age");

// getting name attribute size
eyedb::Size name_length;
attr_name->getSize(o, name_length);

// getting name attribute value
char *name = new char[name_length];
attr_name->getValue(o, (eyedb::Data *)name, name_length, 0);
cout << "name is : " << name << endl;
delete [] name;

// getting age attribute value
eyedb::_int32 age;
attr_age->getValue(o, (eyedb::Data *)&age, 1, 0);
cout << "age is : " << age << endl;
```

To get an attribute value we use the method `eyedb::Attribute::getValue(const eyedb::Object *o, eyedb::Data *data, int nb, int from, eyedb::Bool * isnull = 0)` whose arguments are as follows:

1. `eyedb::Object *o`: the runtime object pointer.
2. `eyedb::Data data`: the pointer to the attribute value to get: this pointer must be allocated correctly according to the returned value type. It is why we get first the size of the `name` attribute value to allocate the returned buffer with a valid size.
3. `int nb`: for an array, the number of values to get.
4. `int from`: for an array, the starting index of the values to get.
5. `eyedb::Bool *isnull`: an optionnal boolean to check if the attribute value is null (i.e. not initialized).

If we want to get the `spouse` value of the loaded person, we must do something a little bit more complicated:

```
eyedb::Oid oid("1456.3.38475637:oid");
eyedb::Object *o;
db.loadObject(oid, o);

// getting spouse attribute from class
const eyedb::Attribute *attr_spouse = cls->getAttribute("spouse");

eyedb::Oid spouse_oid;
attr_spouse->getOid(o, &spouse_oid);

if (spouse_oid.isValid()) {
  eyedb::Object *spouse;
  db.loadObject(spouse_oid, spouse);
  cout << "spouse is: " << spouse;
}
```

To get the `spouse` attribute value, we need to get first the `spouse OID` using the `eyedb::Attribute::getOid` method on the `spouse` attribute.
Then, if the `OID` is valid, we load the `spouse` from the found `OID`.

Once again, using the `eyedbodl` tool, all the previous code becomes very simple:

```
eyedb::Oid oid("1456.3.38475637:oid");
eyedb::Object *o;
db.loadObject(oid, o);
Person *p = Person_c(o);

cout << "name is : " << p->getName() << endl;
cout << "age is : " << p->getAge() << endl;

cout << "spouse is: " << p->getSpouse();
```

1.12 Loading Database Objects using OQL

We have seen in the previous section how to load a database object from its `OID`. The problem is that the `OID` is a rather hidden concept and there are very few chances to know an object `OID` before having loaded it.

To load database objects it seems more reasonnable to use a query language such as OQL.
The EyeDB C++ API allows you to perform any OQL queries using the class `eyedb::OQL`.
For instance to get all Person whose age is less than a given value:

```
db.transactionBegin();
eyedb::OQL q(&db, "select Person.age < %d", given_age);

eyedb::ObjectArray obj_arr(eyedb::True);
q.execute(obj_arr);
for (int i = 0; i < obj_arr.getCount(); i++)
  cout << obj_arr[i];
```

A few remarks about this code:

1. remember that any persistent runtime object manipulation must be done in the scope of a transaction: it is why the first statement is a call to the `transactionBegin` method. In most of the previous code examples, we volontary omit this call.

2. the class `eyedb::OQL` is used to perform any OQL query. The main constructor `eyedb::OQL(eyedb::Database *db, const char *fmt, ...)` allows you to make an OQL query in a simple way. The arguments are as follows:

   (a) the database pointer within which to perform the query.

   (b) the format of the query in a `sprintf` style.

   (c) the other arguments are the arguments related to the previous format.

3. to get all the objects returned by the query, we use the `eyedb::OQL::execute(eyedb::ObjectArray &)` method. This method filled the object array reference given as input parameter.

4. the method `eyedb::ObjectArray::getCount()` returned the number of objects contained in an object array.

5. the `[] eyedb::ObjectArray` operator has been overloaded to allow you to perform direct access to the contained objects: `obj_arr[i]` is the object #`i` within the object array.

6. the argument `eyedb::True` to the `eyedb::ObjectArray` constructor means that we want that all the contained objects to be deleted when this object array will be deleted.

Sometimes we want to perform a query to get only a part of some objects.
For instance, to get the name of all persons whose age is less than a given value, there are two ways:

1. the first one is to get all the persons whose age is less than the given value using an OQL query, and then get their name value as follows:

```
eyedb::OQL q(&db, "select Person.age < %d", given_age);
eyedb::ObjectArray obj_arr(eyedb::True);
q.execute(obj_arr);

const eyedb::Attribute *attr_name = cls->getAttribute("name");
for (int i = 0; i < obj_arr.getCount(); i++) {
   // getting name attribute size
  eyedb::Size name_length;
  attr_name->getSize(obj_arr[i], name_length);

  // getting name attribute value
  char *name = new char[name_length];
  attr_name->getValue(obj_arr[i], (eyedb::Data *)name, name_length, 0);
  cout << "name of #" << i << " is : " << name << endl;
  delete [] name;
}
```

2. the second one is to perform directly an appropriate query as follows:

```
eyedb::OQL q(&db, "(select Person.age < %d).name", given_age);
eyedb::ValueArray val_arr;

q.execute(val_arr);
for (int i = 0; i < val_arr.getCount(); i++)
  cout << "name of #" << i << " is : " << val_arr[i].str << endl;
```

   In this case, the returned value are not object values but string values. So we cannot use the `execute(eyedb::ObjectArray&)` method to get these values but the more general form `execute(eyedb::ValueArray&)`

   An `eyedb::ValueArray` instance is an array of `eyedb::Value` instances. The `eyedb::Value` class is the most general form of an OQL returned value. It can take the form of a integer, a string, an `OID`, an object and so on.

   Note that this second way is more efficient as only the person name are returned from the server and not the full object.

1.13 Releasing Runtime Objects

All the runtime objects which have been allocated by the client code or by a load or query method must be released by the client code.

To release an `eyedb::Object` or inherited class instance, we must use the `eyedb::Object::release()` method as follows:

```
eyedb::Object *o1 = cls->newObj();
// ...
o1->release();

eyedb::Object *o2;
db.loadObject(oid, o2);
// ...
o2->release();
```

The C++ `delete` operator is forbiden: if you try to use this operator on any `eyedb::Object` instance, you will get an error message at runtime.

Note that if you release a persistent runtime object you do not release the tied database object.

Refer to the section `Memory Management` to understand the whole memory policy of the C++ API.

1.14 Removing Database Objects

To remove a database object, we need to use the `eyedb::Object::remove()` method or the `eyedb::Database::removeObject(const eyedb::Oid &oid)` method, for instance:

```
db.transactionBegin();

o->remove();
o->release();

db.transactionCommit();
```

or:

```
db.transactionBegin();

db.removeObject(oid);

db.transactionCommit();
```

When calling one of the previous remove methods, it is not necessary to call the store method after.

2 The Schema-Oriented Generated C++ API

The generic C++ API allows you to manipulate any object within any database: this is its force. But, as shown in the previous section, object manipulation is sometimes very heavy as the provided methods are too much generic.

To enrich the generic API, one introduces a tool to generate specific C++ code from a specific ODL schema: the generated API is call a schema-oriented API.

The schema-oriented API contains mainly:

1. a C++ class for each class defined in the ODL schema.
2. selector and modifier methods in the C++ class for each attribute defined in the ODL class.
3. user friendly selector and modifier methods for array and collection attributes.
4. a C++ method mapped on each method defined in the ODL class.
5. a specific C++ database class used to open a database and check its schema.
6. some utilities such as down-casting funtions.

The schema-oriented API is designed so that the object manipulation for this schema is the most comfortable as possible.

2.1 Generating a Schema-Oriented C++ API

To generate a schema-oriented C++ API, one needs a well formed ODL file describing a schema or a reachable database containing this schema and the `eyedbodl` tool.
To generate a schema-oriented C++ API, the minimal `eyedbodl` invocation is as follows:

```
eyedbodl --gencode=C++ <odlfile>
```

or

```
eyedbodl --gencode=C++ --package=<package> -d <database>
```

For instance, to generate the schema-oriented C++ API for the `person.odl` schema:

```
eyedbodl --gencode=C++ person.odl
```

For a given *package*.odl ODL file, the generated files are as follows:

- *package*.h, *package*.cc: the generated C++ API to be used in a client program
- template_*package*.cc: an example of a client program using the generated API
- Makefile.*package*: an example of Makefile to compile *package*.cc and template_*package*.cc: make -f Makefile.*package* will compile and link the generated API and template files
- *package*stubsfe.cc, *package*stubsbe.cc: stubs for client and server methods
- *package*mthfe-skel.cc, *package*mthbe-skel.cc: skeletons for client and server methods

The `eyedbodl` tool contains a lot of command line options to control the generated code.

There is one mandatory option:
odlfile`|-|-d` *dbname*`|--database=`*dbname* : Input ODL file (or - for standard input) or the database name
and some optionnal options:

`--package=`*package*	: Package name
`--output-dir=`*dirname*	: Output directory for generated files
`--output-file-prefix=`*prefix*	: Ouput file prefix (default is the package name)
`--class-prefix=`*prefix*	: Prefix to be put at the begining of each runtime class
`--db-class-prefix=`*prefix*	: Prefix to be put at the begining of each database class
`--attr-style=`implicit	: Attribute methods have the attribute name
`--attr-style=`explicit	: Attribute methods have the attribute name prefixed by get/set (default)
`--schema-name=`*schname*	: Schema name (default is *package*)
`--export`	: Export class instances in the .h file
`--dynamic-attr`	: Uses a dynamic fetch for attributes in the get and set methods
`--down-casting=yes`	: Generates the down casting methods (the default)
`--down-casting=no`	: Does not generate the down casting methods
`--attr-cache=yes`	: Use a second level cache for attribute value
`--attr-cache=no`	: Does not use a second level cache for attribute value (the default)
`--namespace=`*namespace*	: Define classes with the namespace *namespace*
`--c-suffix=`*suffix*	: Use *suffix* as the C file suffix
`--h-suffix=`*suffix*	: Use *suffix* as the H file suffix
`--gen-class-stubs`	: Generates a file class_stubs.h for each class
`--class-enums=yes`	: Generates enums within a class
`--class-enums=no`	: Do not generate enums within a class (default)
`--gencode-error-policy=status`	: Status oriented error policy (the default)
`--gencode-error-policy=exception`	: Exception oriented error policy
`--rootclass=`*rootclass*	: Use *rootclass* name for the root class instead of the package name
`--no-rootclass`	: Does not use any root class

For instance to generate a schema-oriented C++ API in the directory `tmp`, prefixing the runtime classes with `pp`, suffixing C++ files with `.cpp`, we invoke `eyedbodl` as follows:

```
eyedbodl --gencode C++ --output-dir=tmp --class-prefix=pp \
         --c-suffix=.cpp person.odl
```

2.2 The Generated Code

Seven files are generated:

1. the header C++ file: *package*`.h` (for instance `person.h`)
2. the core C++ file: *package*`.cc` (for instance `person.cc`)
3. files for frontend and backend user method support:
   (a) stubs: *package*`stubsfe.h` and *package*`stubsbe.h`
   (b) skeleton: *package*`mthfe-skel.h` and *package*`mthbe-skel.h`
   (c) a template Makefile: `Makefile.`*package*

   (for instance `Makefile.person`)

The use of the generated files for the user methods are introduced in the chapter `Methods and Triggers`.

The header file contains C++ class declarations and function prototypes.
The following classes are generated:

1. the package class whose name is the package name and which contains a static init method, a static release method and two methods for schema update within a database, for instance:

```
class person {
 public:
  static void init();
  static void release();
  static eyedb::Status updateSchema(eyedb::Database *db);
  static eyedb::Status updateSchema(eyedb::Schema *m);
};
```

   (a) the `person::init()` method must be called before any use of the schema-oriented API.

   (b) the `person::release()` should be called after any use of this API, but this call is not mandatory as this method only release allocated runtime memory.

   (c) the `person::updateSchema()` methods are not generally called directly by client code.

2. the database class whose name is *package*Database inherited from the generic `eyedb::Database` class.
   This class overloads two inherited methods: the `open` and the `loadObject_realize` methods.
   The overloaded `open` method has two purposes:

   - database opening.
   - schema checking: it checks that the opened database schema is strictly identical to the runtime schema.

   The `loadObject_realize` method has one purpose:

   - runtime object construction: for any object loaded from the database whose class is one of the generated classes (for instance `Person`, `Car`), it call the generated class constructor. For instance if an object loaded is of class `Person` it will perform a `new Person(db)` to construct correctly the loaded object.

   Note that to use the generated schema-oriented API it is not mandatory to use the generated database class: you can use the genetic `eyedb::Database` class; there is a lot of cases where you will get no trouble. But to avoid any potential trouble, it is strongly recommended to use the generated database class.

3. a root class which is the superclass of all generated classes, except the package and the database classes.
   This class is used to facilitate the down casting process.
   If the command line option `-no-rootclass` is specified, the root class is not generated.
   Unless its name is given using the command line option `-rootclass name`, its name is `Root`.

4. for each ODL class, a C++ class is generated with the same name possibly prefixed by a string if specified by the command line option `-class-prefix`. This class is inherited from the root class.

   The generated class contains the following method families:

   (a) constructors.

   (b) down casting methods.

   (c) selector attribute methods.

   (d) modifier attribute methods.

   (e) methods mapped from ODL backend or frontend methods.

   (f) client stubs.

   (g) the destructor.

2.3 Constructors and Copy Operator

For each C++ class, two constructors and the assignment operator are generated:

```
Person(eyedb::Database * = 0);
Person(const Person& x);

Person& operator=(const Person& x);
```

- The first constructor is used to instantiate transient or persistent objects.
  The following code:

  ```
  Person *p = new Person(&db);
  ```

  does nearly the same things as:

  ```
  eyedb::Class *cls = db.getSchema()->getClass("Person");
  eyedb::Object *o = cls->newObj(&db);
  ```

  The major difference is that in the second case, an `eyedb::Object` instance (in fact an `eyedb::Struct` instance) is created while in the first case an `Person` (which inherits from `eyedb::Struct`) instance is created.
  But in both cases, you can use the instantiated object to set and get `Person` attribute values and to synchronize the runtime object with the database.
  To set or get attribute values in the second case, you need to use the `eyedb::Attribute::setValue` or `eyedb::Attribute::getValue` methods while in the first case, you may use the generated selector and modifier methods such as `Person::setName` or `Person::getAge`.

- The second constructor is the copy constructor. For instance:

  ```
  Person *p1 = new Person(&db);

  Person p2 = *p1;
  ```

- At last, the assignment operator can be used as follows:

  ```
  Person *p1 = new Person(&db);
  Person *p2 = new Person(&db);

  *p2 = *p1;
  ```

2.4 Down Casting Methods and Functions

Unless the command line option `-down-casting no` has been used, down casting methods and functions have been generated.
For instance, the following methods have been generated for the `Person` class:

```
class Person : public Root {
  // ...
  virtual Person *asPerson() {return this;}
  virtual const Person *asPerson() const {return this;}
  virtual Employee *asEmployee() {return (Employee *)0;}
  virtual const Employee *asEmployee() const {return (const Employee *)0;}
  // ...
};
```

These methods are very useful to process safe down casting. The down casting may be used in several cases. For instance, if you instantiate an `Employee` object as follows:

```
extern void display(Person *);

Employee *empl = new Employee(&db);
display(empl);
```

the display function expects a `Person` instance: when calling it with an `Employee` instance, we do not make any mistake as the `Employee` class inherits from the `Person` class.
Assume now, that the display function displays the name and the age of the `Person` instance and its salary if the instance is an employee. Using the down casting method `Person::asEmployee()`, one can do as follows:

```
void display(Person *p)
{
  cout << "name : " << p->getName() << endl;
  cout << "age : " << p->getAge() << endl;
  if (p->asEmployee())
    cout << "salary : " << p->asEmployee()->getSalary() << endl;
```

Note that the call to this down casting method cost nearly nothing. Without the help of the down casting method, the previous code becomes:

```
void display(Person *p)
{
  cout << "name : " << p->getName() << endl;
  cout << "age : " << p->getAge() << endl;
  if (!strcmp(p->getClass()->getName(), "Employee"))
    cout << "salary : " << ((Employee *)p)->getSalary() << endl;
```

which is rather less efficient and less elegant.

There is another case to use down casting methods and functions is when loading a database object.
When loading a database object (for instance a `Person` database object) using the `eyedb::Database::loadObject`, we get a generic `eyedb::Object` instance, not a `Person` instance nor a `Employee` instance.
Nevertheless, in the case of a `Person` database object has been loaded, a `Person` persistent runtime object has been correctly constructed by the generated API.
So, it is legitimate to down cast the loaded `eyedb::Object` instance to a `Person` instance as follows:

```
  eyedb::OQL q(&db, "select Person.age < %d", given_age);

  eyedb::ObjectArray obj_arr(eyedb::True);
  q.execute(obj_arr);
  for (int i = 0; i < obj_arr.getCount(); i++)
   {
     Person *p = (Person *)obj_arr[i];
     cout << "name: " << p->getName() << endl;
   }
```

The cast:

```
     Person *p = (Person *)obj_arr[i];
```

is legal according to the context but is not safe because neither static (i.e. compiler level) check nor runtime check is performed.
Safe down casting functions are generated by `eyedbodl` as follows:

```
inline Person *Person_c(eyedb::Object *o)
{
  Root *x = personDatabase::asRoot(o);
  if (!x) return (Person *)0;
  return x->asPerson();
}
```

This function allows you to perform compiler and runtime check as follows:

```
  for (int i = 0; i < obj_arr.getCount(); i++)
   {
     Person *p = Person_c(obj_arr[i]);
     if (p)
       cout << "name: " << p->getName() << endl;
   }
```

in the case of the loaded object is not a real `Person` instance, the `Person_c` function returns a null pointer.

It is strongly recommended to make use of these safe down casting methods and functions instead of performing manual down casting.

2.5 Selector Methods

For each attribute in the ODL class, `eyedbodl` generates one or more selector methods.
The number and the form of the selector methods depends on the type of the attribute.
An attribute type is a combination of:

1. a primitive type which can take the form of a:
   (a) basic type: for instance `int32`, `char` or `double`.
   (b) system type: for instance `class`, `object`, `image`.
   (c) user type: for instance `Person`, `Employee`, `set<Car *>`.
   (d) user enum: for instance `CivilState`.
2. the literal or object property:
   (a) the literal property means that the attribute value has no identifier (i.e. `OID`).
   (b) the object property means that the attribute value has an identifier.
3. an optional array modifier:
   (a) multi-dimensionnal and variable size array are supported.

For instance, the attribute:

```
attribute Address addr;
```

can be described as {primitive type = `Address`, property = `literal`, array = `nil`}

The form of the selector methods are designed according to the following attribute type family:

1. **literal basic or user enum type** : `int32 age`, `CivilState cstate`.
2. **literal string** : `string<32> town`, `string name`, `string country`.
3. **literal user type** : `Address addr`
4. **object basic, user or system type** : `Person *spouse`
5. **object collection type** : `array<Person *> children`, `set<Car *>> cars`.

All those type families support in an orthogonal way an multi-dimenstion array modifier.

Literal Basic or User Enum Type

The selector method is under the form:

```
<attribute primitive type> get<attribute name>(eyedb::Bool *isnull = 0,
                                               eyedb::Status *status = 0) const
```

for instance for the `age` attribute:

```
eyedb::_int32 getAge(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
```

Every selector method has the two following optional arguments:

1. `isnull` : a pointer to a `eyedb::Bool` value.
   If this pointer is not null, the selector method assigns it to `eyedb::False` if the attribute value is not null, otherwise it assigns it to `eyedb::True`.
2. `status` : a pointer to a `eyedb::Status` value.
   If this pointer is not null, the selector method assigns to `eyedb::Success` is the operation is successul, otherwise is assigns to the error status. Note that if you are using the exception error policy (the recommended one), this argument is not useful. If you have generated the schema-oriented C++ API using the `-error-policy exception` option, the `status` argument will not be generated.

Literal String

The selector methods are under the form:

```
const char *get<attribute name>(eyedb::Bool *isnull = 0,
                                eyedb::Status *status = 0) const
char get<attribute name>(unsigned int a0,
                         eyedb::Bool *isnull = 0,
                         eyedb::Status *status = 0) const
```

for instance for the `name` attribute:

```
const char *getName(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
char getName(unsigned int a0, eyedb::Bool *isnull = 0, eyedb::Status *status = 0)  const;
```

The first selector method is to get the full string value of the attribute while the second one is to get a specific character within this string value. The argument `a0` is the number of the character.

Literal User Type

The selector methods are under the form:

```
<attribute primitive type>* get<attribute name>(eyedb::Bool *isnull = 0,
                                                eyedb::Status *status = 0)
const <attribute primitive type>* get<attribute name>(eyedb::Bool *isnull = 0,
                                                       eyedb::Status *status = 0) const
```

for instance for the `addr` attribute:

```
Address *getAddr(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) ;
const Address *getAddr(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
```

Note than the returned value cannot be a null pointer as this is literal attribute fully included in the instance.

Object Basic, User or System Type

The selector methods are under the form:

```
<attribute primitive type>* get<attribute name>(eyedb::Bool *isnull = 0,
                                                eyedb::Status *status = 0)
const <attribute primitive type>* get<attribute name>(eyedb::Bool *isnull = 0,
                                                       eyedb::Status *status = 0) const
```

for instance for the `spouse` attribute:

```
Person *getSpouse(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) ;
const Person *getSpouse(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
eyedb::Oid getSpouseOid(eyedb::Status *status = 0);
```

Note that:

1. the returned value can be a null pointer as this a is an object attribute with its own life.
2. for this same reason, there is a method to get the identifier of this object without loading it.
3. this selector method automatically loads the related object attribute when called.

Object Collection Type

As introduced in previous chapter, there are two main types of collections: ordered (or indexed) collections - `array` and `list` - and not ordered collections - `set` and `bag`. The generated methods for these two main types are similar but a little bit different.
For the not ordered collections, the selector method are as follows:

```
<collection type>* get<attribute name>Coll(eyedb::Bool *isnull = 0,
                                           eyedb::Status *status = 0) ;
unsigned int get<attribute name>Count(eyedb::Bool *isnull = 0,
                                      eyedb::Status *status = 0) const;
const <collection type>* get<attribute name>Coll(eyedb::Bool *isnull = 0,
                                                 eyedb::Status *status = 0) const;
```

```
eyedb::Oid get<attribute name>Oid(eyedb::Status *status = 0);
const <collection object type>* get<attribute name>At(unsigned int ind,
                                                       eyedb::Bool *isnull = 0,
                                                       eyedb::Status *status = 0) const;
<collection object type>* get<attribute name>At(unsigned int ind,
                                                eyedb::Bool *isnull = 0,
                                                eyedb::Status *status = 0);
eyedb::Oid get<attribute name>OidAt(unsigned int ind,
                                    eyedb::Status *status = 0) const;
```

where `<collection type>` can be:

1. `eyedb::CollSet` for a collection set
2. `eyedb::CollBag` for a collection bag

and where `<collection object type>` is the type which is composing the collection.

Note that if the collection is not a literal but an object, the following extra method returning the collection oid is generated:

```
eyedb::Oid get<attribute name>Oid(eyedb::Status *status = 0);
```

For the `cars` attribute the following code is generated:

```
eyedb::CollSet *getCarsColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) ;
unsigned int getCarsCount(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
const eyedb::CollSet *getCarsColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
const Car *getCarsAt(unsigned int ind, eyedb::Bool *isnull = 0,
                     eyedb::Status *status = 0) const;
Car *getCarsAt(unsigned int ind, eyedb::Bool *isnull = 0, eyedb::Status *status = 0);
eyedb::Oid getCarsOidAt(unsigned int ind, eyedb::Status *status = 0) const;
```

Let have a look to each method:

1. ```
   eyedb::CollSet *getCarsColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) ;
   const eyedb::CollSet *getCarsColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
   ```
   returns the collection object (const and not const methods).
2. ```
   unsigned int getCarsCount(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
   ```
   returns the collection item count.
3. ```
   const Car *getCarsAt(unsigned int ind, eyedb::Bool *isnull = 0,
                        eyedb::Status *status = 0) const;
   Car *getCarsAt(unsigned int ind, eyedb::Bool *isnull = 0, eyedb::Status *status = 0);
   ```
   returns the #`ind` element in the collection. As the collection is not ordered, the index of the element to get depends on the load ordering and is not specified. These array-oriented methods are generated for user convenience because it is somewhat easier to scan an array that to scan an unordered set.
4. ```
   eyedb::Oid getCarsOidAt(unsigned int ind, eyedb::Status *status = 0) const;
   ```
   returns the #`ind` oid in the collection. The remark about the index of the element is the same as above.

For the ordered collections, the selector method are as follows:

```
<collection type>* get<attribute name>Coll(eyedb::Bool *isnull = 0,
                                           eyedb::Status *status = 0) ;
unsigned int get<attribute name>Count(eyedb::Bool *isnull = 0,
                                      eyedb::Status *status = 0) const;
const <collection type>* get<attribute name>Coll(eyedb::Bool *isnull = 0,
                                                 eyedb::Status *status = 0) const;
eyedb::Oid get<attribute name>Oid(eyedb::Status *status = 0);
const <collection object type>* retrieve<attribute name>At(unsigned int ind,
                                                           eyedb::Bool *isnull = 0,
                                                           eyedb::Status *status = 0) const;
<collection object type>* retrieve<attribute name>At(unsigned int ind,
                                                     eyedb::Bool *isnull = 0,
                                                     eyedb::Status *status = 0);
eyedb::Oid retrieve<attribute name>OidAt(unsigned int ind,
                                         eyedb::Status *status = 0) const;
```

where `<collection type>` can be:

1. `eyedb::CollArray` for a collection array
2. `eyedb::CollList` for a collection list
   Note that the collection list are currently not implemented in EyeDB .

and where `<collection object type>` is the type which is composing the collection.

Note that if the collection is not a literal but an object, the following extra method returning the collection oid is generated:

```
eyedb::Oid get<attribute name>Oid(eyedb::Status *status = 0);
```

For the `children` attribute the following code is generated:

```
eyedb::CollArray *getChildrenColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) ;
unsigned int getChildrenCount(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
const eyedb::CollArray *getChildrenColl(eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
const Person *retrieveChildrenAt(unsigned int ind, eyedb::Bool *isnull = 0,
                                 eyedb::Status *status = 0) const;
Person *retrieveChildrenAt(unsigned int ind, eyedb::Bool *isnull = 0,
                           eyedb::Status *status = 0);
eyedb::Oid retrievedChildrenOidAt(unsigned int ind, eyedb::Status *status = 0) const;
```

Only the last three method templates differ from the corresponding `Car` method templates:

1. ```
   const Person *retrievedChildrenAt(unsigned int ind, eyedb::Bool *isnull = 0,
                                     eyedb::Status *status = 0) const;
   Person *retrievedChildrenAt(unsigned int ind, eyedb::Bool *isnull = 0,
                               eyedb::Status *status = 0);
   ```
   returns the #`ind` element in the collection. As this collection is ordered, the index of the element to get is fully pertinent.
2. ```
   eyedb::Oid retrievedChildrenOidAt(unsigned int ind, eyedb::Status *status = 0) const;
   ```
   returns the #`ind` oid in the collection. As this collection is ordered, the index of the element to get is fully pertinent.

Array Modifier

When an array modifier is present for an attribute, all the previous selector methods change in the same way: for each dimension in the array, an index argument is added at the begining of the selector method.
For instance, for an attribute `int x[23][12]`, the selector methods becomes:

```
eyedb::_int32 getX(unsigned int a0, unsigned int a1,
                   eyedb::Bool *isnull = 0, eyedb::Status *status = 0) const;
```

A call to `getX(1, 3)` returns the attribute value `x[1][3]`.
If the left dimension is variable, for instance `int x[][12]`, the following extra method is generated:

```
unsigned int getXCount(eyedb::Status * = 0) const;
```

For instance, for the `other_addrs` literal user type attribute, the following code is generated:

```
Address *getOtherAddrs(unsigned int a0, eyedb::Bool *isnull = 0, eyedb::Status * = 0) ;
const Address *getOtherAddrs(unsigned int a0, eyedb::Bool *isnull = 0,
                             eyedb::Status * = 0) const;
unsigned int getOtherAddrsCount(eyedb::Status * = 0) const;
```

2.6 Modidier Methods

The modifier methods are very similar to the selector methods. Their forms and their number depends on the same attribute type characteristics as the modifier methods.

Literal Basic or User Enum Type

The modifier method is under the form:

```
eyedb::Status set<attribute name>(<attribute primitive type>);
```

for instance for the `age` attribute:

```
eyedb::Status setAge(eyedb::_int32);
```

Literal String

The modifier methods are under the form:

```
eyedb::Status set<attribute name>(const char *);
eyedb::Status set<attribute name>(unsigned int a0, char);
```

for instance for the `name` attribute:

```
eyedb::Status setName(const char *);
eyedb::Status setName(unsigned int a0, char);
```

Literal User Type

The modifier methods are under the form:

```
eyedb::Status set<attribute name>(<attribute primitive type>*);
```

for instance for the `addr` attribute:

```
eyedb::Status setAddr(Address*);
```

Object Basic, User or System Type

The modifier methods are under the form:

```
eyedb::Status set<attribute name>(<attribute primitive type>*);
eyedb::Status set<attribute name>Oid(const eyedb::Oid &);
```

for instance for the `spouse` attribute:

```
eyedb::Status setSpouse(Person*);
eyedb::Status setSpouseOid(const eyedb::Oid &);
```

Object Collection Type

For unordered collection types, the modifier methods are under the form:

```
eyedb::Status set<attribute name>Coll(<collection type>*);
eyedb::Status set<attribute name>Oid(const eyedb::Oid &);
eyedb::Status addTo<attribute name>Coll(<collection object type>*,
                                        unsigned int magorder = 0);
eyedb::Status rmvFrom<attribute name>Coll(<collection object type>*);
eyedb::Status addTo<attribute name>Coll(const eyedb::Oid &,
                                        unsigned int magorder = 0);
eyedb::Status rmvFrom<attribute name>Coll(const eyedb::Oid &);
```

where `<collection type>` can be:

1. `eyedb::CollSet` for a collection set
2. `eyedb::CollBag` for a collection bag

and where `<collection object type>` is the type which composing the collection.

Note that if the collection is not a literal but an object, the following extra method setting the collection oid is generated:

```
eyedb::Status set<attribute name>Oid(const eyedb::Oid &);
```

For the `cars` attribute, the following code is generated:

```
eyedb::Status setCarsColl(eyedb::CollSet*);
eyedb::Status addToCarsColl(Car*, unsigned int magorder = 0);
eyedb::Status addToCarsColl(const eyedb::Oid &, unsigned int magorder = 0);
eyedb::Status rmvFromCarsColl(Car*);
eyedb::Status rmvFromCarsColl(const eyedb::Oid &);
```

Let have a look to each method:

1. `eyedb::Status setCarsColl(eyedb::CollSet *coll);`

   sets the `cars` attribute collection to the input argument `coll`.

2. `eyedb::Status addToCarsColl(Car *car, unsigned int magorder = 0);`

   adds the `car` instance to the collection attribute `cars`. If the collection is not yet created, this method call will create one using the `magorder` argument for its magnitude order value.

3. `eyedb::Status addToCarsColl(const eyedb::Oid &car_oid, unsigned int magorder = 0);`

   adds the instance of `Car` whose oid is `car_oid` to the collection attribute `cars`. If the collection is not yet created, this method call will create one using the `magorder` argument for its magnitude order value.

4. `eyedb::Status rmvFromCarsColl(Car *car);`

   removes the `car` instance from the collection attribute `cars`. If the instance is not found, an error is raised.

5. `eyedb::Status rmvFromCarsColl(const eyedb::Oid &car_oid);`

   removes the instance of `Car` whose oid is `car_oid` from the collection attribute `cars`. If the instance is not found, an error is raised.

For ordered collection types, the modifier methods are under the form:

```
eyedb::Status set<attribute name>Coll(<collection type>*);
eyedb::Status set<attribute name>Oid(const eyedb::Oid &);
eyedb::Status setIn<attribute name>CollAt(int where, <collection object type>*,
                                          unsigned int magorder = 0);
eyedb::Status setIn<attribute name>CollAt(int where, const eyedb::Oid &,
                                          unsigned int magorder = 0);
eyedb::Status unsetIn<attribute name>CollAt(int where);
```

where `<collection type>` can be:

1. `eyedb::CollArray` for a collection array
2. `eyedb::CollList` for a collection list
   Note that the collection list are currently not implemented in EyeDB .

and where `<collection object type>` is the type which composing the collection.

Note that if the collection is not a literal but an object, the following extra method setting the collection oid is generated:

```
eyedb::Status set<attribute name>Oid(const eyedb::Oid &);
```

For the `children` attribute, the following code is generated:

```
eyedb::Status setChildrenColl(eyedb::CollArray*);
eyedb::Status setChildrenOid(const eyedb::Oid &);
eyedb::Status setInChildrenCollAt(int where, Person*, unsigned int magorder = 0);
eyedb::Status unsetInChildrenCollAt(int where);
eyedb::Status setInChildrenCollAt(int where, const eyedb::Oid &,
                                  unsigned int magorder = 0);
```

Let have a look to each method:

1. `eyedb::Status setChildrenColl(eyedb::CollSet *coll);`

   sets the `children` attribute collection to the input argument `coll`.

2. `eyedb::Status setInChildrenCollAt(int where, Person *person,`
   `unsigned int magorder = 0);`

   adds the `person` instance to the collection attribute `children` at position `where`. If the collection is not yet created, this method call will create one using the `magorder` argument for its magnitude order value.

3. `eyedb::Status setInChildrenCollAt(int where, const eyedb::Oid &person_oid,`
   `unsigned int magorder = 0);`

   adds the instance of `Person` whose oid is `person_oid` to the collection attribute `children` at position `where`.. If the collection is not yet created, this method call will create one using the `magorder` argument for its magnitude order value.

4. `eyedb::Status unsetInChildrenCollAt(int where);`

   removes the instance found at position `where` from the collection attribute `children`.

Array Modifier

When an array modifier is present for an attribute, all the previous modifier methods change in the same way: for each dimension in the array, an index argument is added at the begining of the selector method.
For instance, for an attribute `int x[32][64]`, the modifier methods becomes:

```
eyedb::Status setX(unsigned int a0, unsigned int a1, eyedb::_int32);
```

A call to `setX(2, 24)` sets the attribute value `x[2][24]`.

For instance, for the `other_addrs` literal user type attribute, the following code is generated:

```
eyedb::Status setOtherAddrs(unsigned int a0, Address *);
eyedb::Status setOtherAddrsCount(unsigned int count);
```

Methods mapped from ODL methods

For each ODL class method, there is a generated C++ method with the same name and the corresponding type.
The generated methods in our example is as follows:

```
virtual eyedb::Status change_address(const char * street, const char * town,
                                     char * &oldstreet, char * &oldtown,
                                     eyedb::_int32 &retarg);

static eyedb::Status getPersonCount(eyedb::Database *db, eyedb::_int32 &retarg);
```

2.7 Initialization

The minimal EyeDB C++ program using a generated schema-oriented API is as follows (using our example):

```
#include "person.h"

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);
  person::init();
  // ...
  person::release();
  eyedb::release();
  return 0;
}
```

A few remarks about this code:

1. the file `person.h` contains the whole generated C++ API and includes the generic EyeDB API.
2. the EyeDB C++ layer must be initialized using one of the static method `init` method of the class `EyeDB`.
3. the generated C++ layer must be initialized using the static method `init` of the class *package*.
4. the last statements `person::release()` and `eyedb::release()` allow you to release all the allocated resources and to close opened databases and connections.
   Note that this statement is optionnal as all allocated resources, opened databases and connections will be automatically released or closed in the `exit()` function.

2.8 Database Opening

As shown in a previous section, it is recommended to use the generated C++ database class to open a database with the template schema.
For instance:

```
eyedb::Connection conn;
conn.open();

const char *dbname = argv[1];

person::Database db(dbname);
db.open(&conn, eyedb::Database::DBRW);
```

3 Examples

This section introduces a few complete simple examples that can be found in the directory *prefix*`/share/doc/eyedb/examples`. The `README` file describes the way to compile and run these examples.
The first two programs listed here introduce the generic C++ API of EyeDB while the two following programs presents the generated schema-oriented C++ API through the simple schema example introduced in this chapter. The last example shows `EYEDBDBM` instance manipulation.

3.1 Generic Query Example

This example introduces a simple query program which takes two arguments: the database name and an OQL construct. It executes the OQL construct and displays on its standard output the returned atoms.

```
// examples/C++Binding/generic/query/query.cc

#include <eyedb/eyedb.h>

using namespace std;

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);

  if (argc != 3) {
    fprintf(stderr, "usage: %s <dbname> <query>\n", argv[0]);
    return 1;
  }

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);

  try {
    eyedb::Connection conn;
    // connecting to the eyedb server
    conn.open();

    eyedb::Database db(argv[1]);

    // opening database argv[1]
    db.open(&conn, eyedb::Database::DBRW);

    // beginning a transaction
    db.transactionBegin();

    // performing the OQL query argv[2] using the eyedb::OQL interface
    eyedb::OQL q(&db, argv[2]);

    eyedb::ValueArray arr;
    q.execute(arr);

    cout << "###### Performing the OQL query " << argv[2] <<
      " using the eyedb::OQL interface" << endl;

    // for each value returned in the query, display it:
    for (int i = 0; i < arr.getCount(); i++) {
      // in case of the returned value is an oid, load it first:
      if (arr[i].type == eyedb::Value::OID) {
        eyedb::Object *o;
        db.loadObject(arr[i].oid, &o);
        cout << o;
        o->release();
      }
      else
        cout << arr[i] << endl;
```

```
}

// performing the same query using eyedb::OQLIterator interface
// [1]: getting all returned values

cout << "\n###### Performing the same query using eyedb::OQLIterator "
  "interface: getting all returned values" << endl;

eyedb::OQLIterator iter(&db, argv[2]);
eyedb::Value val;

while (iter.next(val)) {
  // in case of the returned value is an oid, load it first:
  if (val.getType() == eyedb::Value::OID) {
    eyedb::Object *o;
    db.loadObject(val.oid, &o);
    cout << o;

    // in case of the returned object is a collection, display its
    // contents
    if (o->asCollection()) {
      eyedb::CollectionIterator citer(o->asCollection());
      cout << "contents BEGIN\n";
      eyedb::Object *co;
      while(citer.next(co)) {
        cout << co;
        co->release();
      }
      cout << "contents END\n\n";
    }
    // in case of the returned object is a class, display its
    // extent
    else if (o->asClass()) {
      eyedb::ClassIterator citer(o->asClass());
      cout << "extent BEGIN\n";
      eyedb::Object *co;
      while(citer.next(co)) {
        cout << co;
        co->release();
      }
      cout << "extent END\n\n";
    }

    o->release();
  }
  else
    cout << val << endl;
}

// [2]: getting only returned objects

cout << "\n###### Performing the same query using eyedb::OQLIterator "
  "interface: getting only returned objects" << endl;

eyedb::OQLIterator iter2(&db, argv[2]);
eyedb::Object *o;

while (iter2.next(o)) {
  cout << o;
  o->release();
}

// committing the transaction
```

```
    db.transactionCommit();
  }

  catch(eyedb::Exception &e) {
    cerr << argv[0] << ": " << e;
    eyedb::release();
    return 1;
  }

  eyedb::release();

  return 0;
}
```

For instance:

```
./query person "select Person"
./query EYEDBDBM "select class"
```

3.2 Generic Storing Example

This example introduces a simple store program which takes three arguments: the database name, a person name and a person age. It creates a new instance of person using the given name and age.

```
// examples/C++Binding/generic/store/store.cc

#include <eyedb/eyedb.h>

using namespace std;

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);

  if (argc != 4) {
    fprintf(stderr, "usage: %s <dbname> <person_name> <person_age>\n",
            argv[0]);
    return 1;
  }

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);

  try {
    eyedb::Connection conn;
    // connecting to the eyedb server
    conn.open();

    eyedb::Database db(argv[1]);

    // opening database argv[1]
    db.open(&conn, eyedb::Database::DBRW);

    // beginning a transaction
    db.transactionBegin();

    // looking for class 'Person'
    eyedb::Class *personClass = db.getSchema()->getClass("Person");

    // looking for the attribute 'name' and 'age' in the class 'Person'
    const eyedb::Attribute *name_attr = personClass->getAttribute("name");
    if (!name_attr) {
      fprintf(stderr, "cannot find name attribute in database\n");
      return 1;
    }
```

```
    const eyedb::Attribute *age_attr = personClass->getAttribute("age");

    if (!age_attr) {
      fprintf(stderr, "cannot find age attribute in database\n");
      return 1;
    }

    // instanciating a 'Person' object
    eyedb::Object *p = personClass->newObj(&db);

    // setting the name argv[2] to the new Person instance
    name_attr->setSize(p, strlen(argv[2])+1);
    name_attr->setValue(p, (eyedb::Data)argv[2], strlen(argv[2])+1, 0);

    // setting the age argv[3] to the new Person instance
    int age = atoi(argv[3]);
    age_attr->setValue(p, (eyedb::Data)&age, 1, 0);
    p->store();

    // committing the transaction
    db.transactionCommit();
  }

  catch(eyedb::Exception &e) {
    cerr << e;
    eyedb::release();
    return 1;
  }

  eyedb::release();

  return 0;
}
```

For instance:

```
./store person john 32
./store person mary 28
```

3.3 Schema-Oriented Query Example

This example introduces a simple schema-oriented query program which takes two arguments: the database name and an OQL construct. It executes the OQL construct and displays on its standard output the returned atoms.

```
// examples/C++Binding/schema-oriented/query/query.cc

#include "person.h"

using namespace std;

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);
  person::init();

  if (argc != 3) {
    fprintf(stderr, "usage: %s <dbname> <query>\n", argv[0]);
    return 1;
  }

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);

  try {
```

```
    eyedb::Connection conn;
    // connecting to the eyedb server
    conn.open();

    // opening database argv[1] using 'personDatabase' class
    personDatabase db(argv[1]);
    db.open(&conn, eyedb::Database::DBRW);

    // beginning a transaction
    db.transactionBegin();

    // performing the OQL query argv[2]
    eyedb::OQL q(&db, argv[2]);

    eyedb::ObjectArray arr;
    q.execute(arr);

    // for each Person returned in the query, display its name and age,
    // its address, its spouse name and age and its cars
    for (int i = 0; i < arr.getCount(); i++) {
      Person *p = Person_c(arr[i]);
      if (p) {
        cout << "name:    " << p->getName() << endl;
        cout << "age:     " << p->getAge() << endl;

        if (p->getAddr()->getStreet().size())
          cout << "street:  " << p->getAddr()->getStreet() << endl;

        if (p->getAddr()->getTown().size())
          cout << "town:    " << p->getAddr()->getTown() << endl;

        if (p->getSpouse()) {
          cout << "spouse_name: " << p->getSpouse()->getName() << endl;
          cout << "spouse_age:  " << p->getSpouse()->getAge() << endl;
        }

        eyedb::CollectionIterator iter(p->getCarsColl());
        Car *car;
        while (iter.next((eyedb::Object *&)car)) {
          cout << "car: #" << i << ": " <<
            car->getBrand() << ";" <<
            car->getNum() << endl;
        }
      }
    }

    // committing the transaction
    db.transactionCommit();
  }

  catch(eyedb::Exception &e) {
    cerr << argv[0] << ": " << e;
    eyedb::release();
    return 1;
  }

  eyedb::release();

  return 0;
}
```

For instance:

```
./query person "select Person"
```

3.4 Schema-Oriented Storing Example

This example introduces a simple schema-oriented store program which takes four arguments: the database name, a person name, a person age and the name of its spouse. It creates a new instance of person using the given name and age and mary this person to the spouse whose name is given.

```
// examples/C++Binding/schema-oriented/store/store.cc

#include "person.h"

int
main(int argc, char *argv[])
{
  // initializing the EyeDB layer
  eyedb::init(argc, argv);

  // initializing the person package
  person::init();

  if (argc != 5) {
    fprintf(stderr, "usage: %s <dbname> <person name> <person age> "
            "<spouse name>\n", argv[0]);
    return 1;
  }

  const char *dbname = argv[1];
  const char *name = argv[2];
  int age = atoi(argv[3]);
  const char *spouse_name = argv[4];

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);

  try {
    eyedb::Connection conn;

    // connecting to the EyeDB server
    conn.open();

    // opening database dbname using 'personDatabase' class
    personDatabase db(dbname);
    db.open(&conn, eyedb::Database::DBRW);

    // beginning a transaction
    db.transactionBegin();

    // first looking for spouse
    eyedb::OQL q(&db, "select Person.name = \"%s\"", spouse_name);

    eyedb::ObjectArray arr;
    q.execute(arr);

    // if not found, returns an error
    if (!arr.getCount()) {
      fprintf(stderr, "cannot find spouse '%s'\n", spouse_name);
      return 1;
    }

    // (safe!) casting returned object
    Person *spouse = Person_c(arr[0]);

    // creating a Person
    Person *p = new Person(&db);

    p->setCstate(Sir);
```

```
    p->setName(name);
    p->setAge(age);

    p->setSpouse(spouse);

    // spouse is no more necessary: releasing it
    spouse->release();

    p->getAddr()->setStreet("voltaire");
    p->getAddr()->setTown("paris");

    // creating two cars
    Car *car1 = new Car(&db);
    car1->setBrand("renault");
    car1->setNum(18374);

    Car *car2 = new Car(&db);
    car2->setBrand("ford");
    car2->setNum(233491);

    // adding the cars to the created person
    p->addToCarsColl(car1);
    p->addToCarsColl(car2);

    // car pointers are no more necessary: releasing them
    car1->release();
    car2->release();

    // creating ten children
    for (int i = 0; i < 10; i++) {
      Person *c = new Person(&db);
      char tmp[64];

      c->setAge(i);
      sprintf( tmp, "%d", i);
      c->setName( (std::string(name) + std::string("_") + std::string(tmp)).c_str() );
      p->setInChildrenCollAt(i, c);
      c->release();
    }

    // storing all in database
    p->store(eyedb::RecMode::FullRecurs);

    // committing the transaction
    db.transactionCommit();

    // releasing p
    p->release();
  }

  catch(eyedb::Exception &e) {
    std::cerr << argv[0] << ": " << e;
    eyedb::release();
    return 1;
  }

  // releasing the EyeDB layer: this is not mandatory
  eyedb::release();

  return 0;
}
```

```
./store person wayne 34
./store person poppins 51
```

3.5 Simple Administration Example

This simple example introduces the way to manipulate objects in the EYEDBDBM database. This program:

1. displays the schema of the EYEDBDBM database,
2. displays the EyeDB user names,
3. for each database, it displays the name, the database file and the user access information.

```
// examples/C++Binding/schema-oriented/admin/admin.cc

#include <eyedb/eyedb.h>

using namespace std;

static const char *
get_string_mode(eyedb::DBAccessMode mode)
{
  if (mode == eyedb::NoDBAccessMode)
    return "eyedb::NoDBAccessMode";
  if (mode == eyedb::ReadDBAccessMode)
    return "eyedb::ReadDBAccessMode";
  if (mode == eyedb::WriteDBAccessMode)
    return "eyedb::WriteDBAccessMode";
  if (mode == eyedb::ExecDBAccessMode)
    return "eyedb::ExecDBAccessMode";
  if (mode == eyedb::ReadWriteDBAccessMode)
    return "eyedb::ReadWriteDBAccessMode";
  if (mode == eyedb::ReadExecDBAccessMode)
    return "eyedb::ReadExecDBAccessMode";
  if (mode == eyedb::ReadWriteExecDBAccessMode)
    return "eyedb::ReadWriteExecDBAccessMode";
  if (mode == eyedb::AdminDBAccessMode)
    return "eyedb::AdminDBAccessMode";

  return NULL;
}

int
main(int argc, char *argv[])
{
  // initializing the eyedb layer
  eyedb::init(argc, argv);

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode);

  try {
    eyedb::Connection conn;

    // connecting to the eyedb server
    conn.open();

    // opening the database EYEDBDBM using 'dbmDataBase' class
    eyedb::DBMDatabase db("EYEDBDBM");
    db.open(&conn, eyedb::Database::DBRead);

    // beginning a transaction
    db.transactionBegin();

    // display the scheme on stdout
    cout << db.getSchema() << endl;

    // looking for all user
    eyedb::OQL q_user(&db, "select User");
```

```
    eyedb::ObjectArray user_arr;
    q_user.execute(user_arr);

    cout << "User List {" << endl;
    for (int i = 0; i < user_arr.getCount(); i++) {
      eyedb::UserEntry *user = (eyedb::UserEntry *)user_arr[i];
      cout << "\t" << user->name() << endl;
    }
    cout << "}\n" << endl;

    // looking for all database entry
    eyedb::OQL q_db(&db, "select eyedb::DBEntry");

    eyedb::ObjectArray db_arr;
    q_db.execute(db_arr);

    cout << "Database List {" << endl;

    for (int i = 0; i < db_arr.getCount(); i++) {
      eyedb::DBEntry *dbentry = (eyedb::DBEntry *)db_arr[i];
      cout << "\t" << dbentry->dbname() << " -> " << dbentry->dbfile() << endl;
      // looking for all user which has any permission on this
      // database
      eyedb::OQL q_useraccess(&db,
                              "select eyedb::DBUserAccess->dbentry->dbname = \"%s\"",
                              dbentry->dbname().c_str());
      eyedb::ObjectArray useraccess_arr;
      q_useraccess.execute(useraccess_arr);
      if (useraccess_arr.getCount()) {
        cout << "\tUser Access {" << endl;
        for (int j = 0; j < useraccess_arr.getCount(); j++) {
          eyedb::DBUserAccess *ua = (eyedb::DBUserAccess *)useraccess_arr[j];
          cout << "\t\t" << ua->user()->name() << " -> " <<
            get_string_mode(ua->mode()) << endl;
        }
        cout << "\t}" << endl;
      }
      cout << endl;
      useraccess_arr.garbage();
    }

    cout << "}" << endl;

    // releasing runtime pointers
    db_arr.garbage();
    user_arr.garbage();
  }

  catch(eyedb::Exception &e) {
    cerr << argv[0] << ": " << e;
    eyedb::release();
    return 1;
  }

  // releasing the eyedb layer: this is not mandatory
  eyedb::release();

  return 0;
}
```