Implementation of SQLite database support in program gama-local

Vaclav Petras
Department of Mapping and Cartography
Faculty of Civil Engineering, Czech Technical University in Prague

Keywords: GNU Gama, adjustment of geodetic networks, programming, C, C++, databases, SQLite, callback functions

Abstract

The program gama-local is a part of GNU Gama project and allows adjustment of local geodetic networks. Before realization of this project the program gama-local supported only XML as an input. I designed and implemented support for the SQLite database and thanks to this extension gama-local can read input data from the SQLite database. This article is focused on the specifics of the use of callback functions in C++ using the native SQLite C/C++ Application Programming Interface. The article provides solution to safe calling of callback functions written in C++. Callback functions are called from C library and C library itself is used by C++ program. Provided solution combines several programing techniques which are described in detail, so this article can serve as a cookbook even for beginner programmers. This project was accomplished within my bachelor thesis.

Introduction

GNU Gama is a library and set of programs for adjustment of geodetic networks. Project is licensed under the GNU GPL and is written in C++ programming language. Its main author and developer is professor Aleš Čepek [1] but it has many other contributors. For numerical solutions of least squares adjustment several numerical algorithms (e.g. Singular Value Decomposition and Gram-Schmidt orthogonalization) can be used in GNU Gama [2].

Program gama-local allows adjustment of local-geodetic networks. My work was to implement the support of reading input data from SQLite 3 database [3].

This paper deals with the specifics of using C library (SQLite) in C++ program (gama-local). These specifics result mainly from different function linkage conventions and different approaches to exception handling in C and C++.

All work described here was done within my bachelor thesis [4] which was also used as the main source for writing this paper.

SQLite and gama-local

Program gama-local is able to process classic geodetic measurements (distances, angles, ...) and also measurements such as vectors and coordinates. Input data can be stored in a Gama specific XML file or in a SQLite 3 database file. The same data are stored in both formats. Only identifiers used to connect data in SQLite database are not in XML file because in XML file the most of relations between objects are represented by aggregation.

Formerly gama-local supported only XML input. During development of QGama (gama-local Graphical User Interface) [5] it was realized that SQLite database can be useful for this GUI application. Its next version will be based on using SQLite. To keep the full compatibility between GUI based QGama and command-line based gama-local it was necessary to support SQLite. The support of SQLite database file as an input is sufficient now because only SQLite input is supported in QGama.

Furthermore, SQLite database provides several advantages for gama-local users. For example, more than one (input) network can be stored in one file which is not possible with current gama-local XML format. Database schema used by gama-local can be also used as a part of larger database since other data (tables and columns) are simply and naturally ignored during processing database file by gama-local. This is not true for Gama (expat based) XML parser which does not ignore additional relations and values (represented by XML attributes or elements).

Generally, both XML and SQLite has advantages and disadvantages. For example XML file is in contrast to SQLite database file human readable and editable. Hence SQLite in GNU Gama is not a replacement for XML input but it is intended to be an alternative for whom XML is not the right option (e.g. they don't have a good library support).

The SQLite support is available in GNU Gama version 1.11 [6]. All code related to SQLite is in class `SqliteReader` and its implementation, so I will often refer to this class or its implementation (e.g. `ReaderData` class).

Database schema used by gama-local was developed separately and can be obtained with GNU Gama distribution. Its description can be found in the latest GNU Gama manual (available with latest GNU Gama version from Git repository [7]).

SQLite C/C++ API

SQLite database has native interface for C and C++ which I will refer as SQLite C/C++ API.

SQLite C/C++ API provides several interfaces. I will focus on the two most important interfaces. The first one, which can be called classic, contains functions for executing SQL statement and for retrieving attribute values from result (in case of `SELECT` statement). The second one relies on callback functions.

Working with classic interface consists of calling prepare functions, execute functions, functions for retrieving attribute values and for finalizing statements. All functions return a return code which has to be checked and in case of an error, an error message should be checked. Resulting code can be very long and the using of classic interface can be tedious and error prone. However, classic interface is flexible enough to enable wrapping interface functions by something more convenient. There are several possibilities. In C++ language RAII (Resource Acquisition Is Initialization) technique can be used. This means that C functions and pointers to `struct`s could be wrapped by C++ class with constructor and destructor. Reporting errors by error codes would be probably replaced by exceptions. In C language some wrapper function or functions can be written.

Actually if you decide to use some wrapper, it is not necessary to write wrapper function on your own because it already exists. It is function `sqlite3_exec` from interface using callback functions. Function `sqlite3_exec` is the only one function in this interface (except functions for opening and closing database and for freeing error message).

Project GNU Gama uses interface using callback functions. It was chosen for implementation because this interface is considered to be the most stable one in terms of changes between versions of SQLite C/C++ API. There were no changes in this interface between the last [8] and the current version of SQLite [9].

Simplified example of using a callback function with library function `sqlite3_exec` is shown bellow.

```
// library function
int sqlite3_exec(int(*pf)(void*, int, char**), void* data) {
    // ...
    int rc = pf(data, /*...*/);
    // ...
}
```

Library function gets pointer to callback function as a parameter `pf`. The callback function is called through pointer `pf`. Object `data` given to function `sqlite3_exec` by pointer to `void` is without any change passed to the callback function. The callback function is invoked for each result row and parameters are attribute values of one row of a SQL query result. A user of SQLite C/C++ API writes the callback function and puts the appropriate code in it.

```
// callback function (user code)
int callback(void* data, int argc, char** argv) {
    // ... get values from argv
}
```

All work is done in the callback function, so once a user of SQLite C/C++ API has it, he can simply call function `sqlite3_exec` and pass the pointer to the callback function and the object represented by pointer `data`.

```
// main program (main user code)
int fun() {
    ReaderData* data = /*...*/
    int rc = sqlite3_exec(callback, data);
}
```

Object represented by pointer `data` can be used in the callback function for storing data from parameters. But first, pointer `data` has to be cast from `void*` to particular type (`ReaderData*` in this case). Generally, any type (class) can be chosen depending on the user needs. Later I will show how it is used to store information about an exception.

Using C and C++ together

Several issues has to be considered when C and C++ are used together. The main issues are dealing with function linkage and exception handling. Function linkage is partially solved by (SQLite) library authors. But linkage of callback functions has to be handled by library users. We have to deal with exception handling only when we really use and need exceptions. GNU Gama uses C++ exceptions extensively. Almost all error states are reported by throwing an exception. Callback functions use GNU Gama objects and functions. Therefore, exception can be thrown in callback function. There is no other option but to deal with exception since design decision about using exceptions in GNU Gama project was already done. Decision was mainly influenced by recommendations from [10].

Functions

Functions in C and C++ have different linkage conventions. Functions written in C and compiled by C compiler can be called from C++ but the function declaration has to specify that function has C linkage. The C++ standard specifies that each C++ implementation shall provide C linkage [11]. The C linkage can be specified by `extern "C"` declaration or by enclosing function declaration with `extern "C"` block:

```
extern "C" int f(int);
extern "C" {
    int g(int);
}
```

Common practice used by libraries (C or C++) is to share header files between C and C++. It is achieved using preprocessor:

```
// lib.h:
#ifdef __cplusplus
extern "C" {
#endif
    void f();
    void g(int);
#ifdef __cplusplus
}
#endif
```

C compiler ignores `extern "C"` block but C++ compiler knows that functions have C linkage and compiler uses this information while linking to library or object file.

Function pointers

The similar rules which apply to functions apply also to function pointers. The standard [11] says: *Two function types with different language linkages are distinct types even if they are otherwise identical.* This means that you have to declare C function pointer type inside `extern "C"` block and handle C function and C++ function pointers separately. This is an example of declaration taken from `SqliteReader` implementation:

```
extern "C" {
    typedef int (*SqliteCallback)(void*, int, char**, char**);
}
```

However, GCC [12] provides implicit conversion between C and C++ function pointers. It is allowed as an implementation extension, however doing conversion without any warning and not allowing overloading on language linkage is considered as a bug [13].

Function visibility

Functions in C and C++ have definitions (function body, function code) and declarations (function signature). Function definitions are globally visible by default but function declarations have local visibility. Declaration is visible from the point of declaration to the end of translation unit (source file with included header files).

It is necessary to provide declaration to use function defined in another translation unit. This is always done by including a particular header file. Note that function declarations can be written by hand since including a header file is textual operation only (but this makes sense only in examples).

To avoid name clashes C++ introduced namespaces. Although there is no such thing as namespace in C language namespaces can be used with declaration `extern "C"` together. So the example above can be rewritten using namespace:

```
// lib.h:
#ifdef __cplusplus
namespace lib {
    extern "C" {
#endif
        void f();
        void g(int);
#ifdef __cplusplus
    }
}
#endif
```

Now if you are using `lib.h` header file with C++ compiler, you have to specify `lib` namespace to access functions `f` and `g`. How this can be used with existing C header files (e.g. with C standard library) is described in [10]. Nevertheless, function `f` and `g` are still C functions. This implies that you can provide another declaration without namespace (but with `extern "C"`) and use functions without specifying namespace. This can lead to errors or at least name clashes.

In many cases it is suitable to hide function definition (which is global by default), so it is not visible from outside a translation unit. This is, for example, the case of callback functions which are mostly part of an implementation and therefore they shouldn't be globally visible. The function hiding is done in C++ by unnamed namespace (sometimes incorrectly referred as anonymous namespace) [11]. An unnamed namespace behaves as common namespace but with unique and unknown name (this is done by C++ compiler).

However, unnamed namespace and `extern "C"` cannot be used together. Function previously defined and declared as `extern "C"` in unnamed namespace can be misused or can break compilation, because unnamed namespace behaves as common namespace and `extern "C"` function declaration without namespace can be provided (e.g. accidentally). Example follows.

```
// file_a.cpp:
// unnamed namespace
namespace {
    // bar_i intended to be defined local in file_a.cpp
    extern "C" int bar_i(int) { return 1; }
}

// file_b.cpp:
// bar_i declared (e.g. accidentally)
extern "C" int bar_i(int);

void test() {
    // bar_i used (without any error or warning)
    bar_i(1);
}
```

The function hiding can be also done by declaring function `static`. This is how the hiding is done in C language so it looks appropriately for `extern "C"` functions. Next paragraph discuss it.

Declaring functions in `extern "C"` block as `static` works in GCC as expected. I haven't succeeded in verifying that combination of `extern "C"` and `static` works generally on all compilers. As a result, this solution wasn't used in `SqliteReader` implementation. Instead all callback functions was prefixed. Function definitions are visible but prefix should prevent from misusing by accident.

Exception handling

Handling error (or exception) states is done in C++ by exception handling mechanism. Exceptions have several advantages. Better separation of error handling code from ordinary code is one of them. Another advantage is that exceptions unlike other techniques force programmer to handle error states. For example, return code can be ignored and if there was an error program stays in undefined state. On the other hand, thrown exception can not be ignored since unhandled exception causes program to crash immediately.

C language has no standard exception handling mechanism therefore callback functions called from C library must not throw exception. So a callback function passed to `sqlite3_exec` function have to catch all exceptions thrown inside its body (or by other functions inside its body). From another point of view, function declared `extern "C"` has to behave as a C function and naturally C function does not throw any exception. We should be aware of the fact mentioned above that the code in function body is C++ code and C++ code can use exceptions without any restriction.

Consequently, the callback function has to catch all exceptions. The whole part of function body where exception can be thrown has to by enclosed in `try` block and the last `catch` block has to be `catch` with ellipsis.

```
int callback() {
    // cannot throw exception
    try {
        // can throw exception
    }
    catch (std::exception& e) {
        // handle exception(s) derived from std::exception
    }
    catch (...) {
        // handle unknown exception(s)
    }
}
```

Catching all possible exceptions is not enough, it is necessary to report error to callback function caller (it is library function `sqlite3_exec` in our case). An error can be reported in several ways, in case of SQLite C/C++ API it is returning non-zero return code. Error state reporting is solved easily but the problem is how to provide information about the caught exception (its type and additional information contained in exception object). There are some solutions like assigning return code values to particular exception types. The robust solution which keeps information about exception requires to implement polymorphic exception handling and will be discussed later.

There is also completely different solution of handling exception when interfacing with C language — to use no exceptions at all. However, we would lose all advantages of using exceptions. The second shortcoming of this solution is that exceptions can be already used in code or library we are using. This is the case of the standard library or GNU Gama project.

Polymorphic exception handling

Polymorphic exception handling requires to implement *cloning* and also similar technique for *rethrowing* exceptions. Both will be described in this section and additional information can be found in [10].

Cloning

Standard copying by copy constructor cannot by used in cases when object is held by pointer or reference to a base class because actual type of object is unknown. Using copy constructor would cause slicing [14].

While handling exceptions, references to base exception class are used and proper copy of exception has to be stored (in `SqliteReader` implementation). Proper copy means that new object has the same set of attribute values as the old one and also new object has the same type as the old one. This is the case when cloning must be used instead of copying by copy constructor. Cloning is made by virtual function `clone` which calls copy constructor. In virtual function actual type of object is known and so the right copy constructor is called. Function `clone` creates new object by calling operator `new` and returns pointer to a new object (the user of function is responsible for freeing allocated memory).

Next example shows implementation and simple usage of cloning.

```
class Base {
public:
    virtual ~Base() { }
    virtual Base* clone() { return new Base(*this); }
    virtual std::string name() { return "Base"; }
};

class Derived : public Base {
public:
    virtual Derived* clone() { return new Derived(*this); }
    virtual std::string name() { return "Derived"; }
};

void print(Base* b) {
    std::cout << "name is " << b->name() << std::endl;
}

void test() {
    Base* d = new Derived();
    Base* b = d->clone(); // creates new object
    print(b); // prints: name is Derived
    delete b;
    delete d;
}
```

There is still danger that we accidentally copy (by copy constructor) the object we have by pointer to base class. This can be avoided by declaring copy constructor in base class protected. The second thing we should avoid is forgetting to implement `clone` function in derived classes. This would lead to creating objects of base class instead of derived one. It is helpful to declare `clone` function in base class pure virtual. Unfortunately, this can be applied only for abstract base class and it ensures implementing of `clone` function only for direct subclasses [14].

The same technique which was used to implement cloning can be used more generally to create new objects with various parameters and not only the copy of current and actual object. This technique can be used even for completely different things such as rethrowing exceptions.

Storing and rethrowing exceptions

Classes `Base` and `Derived` from previous section will be used here as exception classes. Both contain functions for cloning and for getting a type name. Both classes also have public copy constructor automatically created by compiler. Public copy constructor is necessary to allow throwing exceptions (by `throw` statement).

An example of standard exception handling is in the following listing.

```
try {
    throw Derived();
}
catch (Derived& e) {
    std::cout << "Derived exception" << std::endl;
}
catch (Base& e) {
    std::cout << "Base exception" << std::endl;
}
```

An exception is thrown somewhere in `try` block. The thrown exception can by caught by one of the caught exceptions. Commonly accepted rule is to throw exceptions by value and to catch them by reference. The order of catch blocks is important (first derived classes then base classes).

Now consider the case we have to catch all exceptions, keep information about exception type and later use this exception. This is the case of callback functions used with `SqliteReader` class. We can create copy of caught exception and store a copy (e.g. in `SqliteReader` class attribute) and later (when we can control program flow) we can pick stored exception up. However, the copy cannot be created by copy constructor because actual type of object is not known. According to previous section, obvious solution is cloning.

Cloning will ensure correct storing of exception by pointer to base class. Pointer to base class allows to use functions from base class interface, for example read error message or get class name in our `Base-Derived` example. However, sometimes final handling of exception and reporting error to program user is not the right thing to do. If it is not clear how to handle an exception, the exception should be thrown again or better say rethrown.

If you try to throw exception directly by `throw` statement using pointer to base class, you will fail because `throw` statement uses copy constructor and known type of object is determined by a pointer type. The pointer type is pointer to base class. Therefore `throw` statement will call base class copy constructor and it will slice the object. Related code snippets can look like this:

```
Base* b = 0;
// ...
b = caughtException.clone() // cloning of exception somewhere
// ...
```

```
throw *b; // rethrowing of exception
```

Direct use of `throw` statement discards useful information.

Slicing while rethrowing can be avoided in the same way as slicing while copying. Polymorphic rethrowing or simply polymorphic throwing has to be introduced. This will be provided by function `raise`. This function is very similar to `clone` function but instead of creating new object it throws an exception. The name `raise` is more appropriate than the name `rethrow` because function can be used more generally than only for rethrowing (the name `throw` cannot be used because it is C++ reserved keyword). The function implementation is the same for all classes and is shown bellow.

```
virtual void raise() const { throw *this; }
```

An exception thrown by this function will has appropriate type because appropriate copy constructor will be used. A usage of this function is obvious and is shown in following code snippet.

```
b->raise();
```

Implementation in SqliteReader class

The exception hierarchy used in `SqliteReader` class is in the source code listing below. Abstract class `Exception::base` provides interface for cloning and rethrowing exceptions (functions `clone` and `raise`). Class is derived from `std::exception` in order to add function `what` to interface and to allow handling standard exceptions and specific GNU Gama exceptions together when necessary. There are many other exceptions in GNU Gama but they are defined and handled in the same way.

```
// inderr.h:
namespace Exception {
    class base: public std::exception
    {
    public:
        virtual base* clone() const = 0;
        virtual void  raise() const = 0;
    };
}

// exception.h:
namespace Exception {
    class string: public base {
    public:
        const std::string  str;
        string(const std::string& s) : str(s) { }
        ~string() throw() {}
        string*     clone() const { return new string(*this); }
        void        raise() const { throw *this; }
        const char* what()  const throw() { return str.c_str(); }
    };
}

// sqlitereader.h:
namespace Exception {
    class sqlitexc: public string
    {
    public:
        sqlitexc(const std::string& message) : string(message) { }
        sqlitexc* clone() const { return new sqlitexc(*this); }
        void      raise() const { throw *this; }
    };
}
```

The next source code listing shows callback function code which is common for all callback functions in `SqliteReader` implementation. All callback functions are declared `extern "C"` and have name prefixed with `sqlite_db_` (as explained in previous sections). The code specific for each callback function is placed in the `try` block. This code can throw any exception. Exceptions derived from `Exception::base` will be caught by first `catch` block, then cloned and stored. Exceptions derived only from `std::exception` will be caught by second `catch` block. Class `std::exception` doesn't allow cloning. Therefore this exception will be replaced by `Exception::string` with same `what` message and then stored. Shortcoming is discarding exception type. Some improvements can be done by introducing new exception to GNU Gama exception hierarchy, e.g. `Exception::std_exception` but it is not such a big improvement because it would save only the information that exception was derived from `std::exception` and actual type of exception would be still unknown. This has the same effect as adding some string to the `what` message. The last `catch` block (`catch` block with ellipsis) ensures that all other exceptions are caught in callback function body. There is no other way than to store exception which indicates unknown exception.

```
extern "C" int sqlite_db_readSomething(void* data, int argc, char** argv)
{
    ReaderData* d = static_cast<ReaderData*>(data);
    try {
        // ... callback's own code
        return 0;
    }
    catch (Exception::base& e) {
        d->exception = e.clone();
    }
    catch (std::exception& e) {
        d->exception = new Exception::string(e.what());
    }
    catch (...) {
        d->exception = new Exception::string("unknown");
    }
  return 1;
}
```

The last source code listing also shows how C++ objects are passed through C to C++ functions. This is common problem with simple solution. Pointer to `void` given to C function (and than to callback function) is casted to appropriate type by `static_cast`. Then C++ object can be used as usual.

In `SqliteReader` implementation a wrapper function for `sqlite3_exec` function was introduced. This wrapper ensures rethrowing of previously stored exception.

```
void exec(sqlite3* sqlite3Handle, const std::string& query,
          SqliteReaderCallbackType callback,
          ReaderData* readerData)
{
    char* errorMsg = 0;
    int rc = sqlite3_exec(sqlite3Handle, query.c_str(), callback,
                          readerData, &errorMsg);
    if (rc != SQLITE_OK) {
        if (readerData->exception != 0) {
            readerData->exception->raise();
        }
        // ... handle other (non-callback) errors
    }
}
```

A `clone` function creates new instances by calling `new` operator as well as it is done in `catch` blocks in callback function. Therefore allocated memory has to be freed. Deallocation can be done easily in `SqliteReader` destructor.

```
SqliteReader::~SqliteReader()
{
    // ...
    if (readerData->exception) {
       delete readerData->exception;
       readerData->exception = 0;
    }
    // ...
}
```

Conclusion

Using C and C++ together

Information provided in this paper was used for implementing SQLite support in gama-local. However, this paper presents something like reusable design pattern because this information can be used generally when interfacing C++ code with C code or library.

The complete source code of `SqliteReader` class and its implementation can be found in GNU Gama Git source code repository [15].

GNU Gama

SQLite database file is an alternative to input XML file. GNU Gama users can now choose format which is appropriate for their project. Users will also be able to switch from GUI application QGama to command-line tool gama-local and back as needed because both programs has the same native format. We will see in the future whether SQLite database file become the main GNU Gama format.

Program gama-local from GNU Gama release 1.11 [6] is able to read adjustment data from SQLite 3 database. User documentation and the latest GNU Gama version are available in the source code repository [7].

References

1. ČEPEK, Aleš. *GNU Gama manual* [online]. 2011-08-16. http://www.gnu.org/software/gama/manual/gama.pdf

2. ČEPEK, Aleš; PYTEL, Jan. *A Note on Numerical Solutions of Least Squares Adjustment in GNU Project Gama* In: Interfacing Geostatistics and GIS. Berlin: Springer-Verlag, 2009, p. 179. ISBN 978-3-540-33235-0.

3. *SQLite* [program]. Version 3. 2004-2011. http://www.sqlite.org/

4. PETRÁŠ, Václav. *Podpora databáze SQLite pro program gama-local.* Praha 2011. 75 s. Bakalářská práce. ČVUT v Praze, Fakulta stavební

http://geo.fsv.cvut.cz/proj/bp/2011/vaclav-petras-bp-2011.pdf

5. NOVÁK, Jiří. *Object – oriented GUI for GNU Gama.* Prague, 2010. 63 p. Bachelor thesis. CTU in Prague, Faculty of Civil Engineering.
http://geo.fsv.cvut.cz/proj/bp/2010/jiri-novak-bp-2010.pdf
6. GNU FTP server mirrors. *GNU Gama release 1.11*
http://ftp.sh.cvut.cz/MIRRORS/gnu/pub/gnu/gama/gama-1.11.tar.gz
7. GNU Gama Git source code repository http://git.savannah.gnu.org/cgit/gama.git/
8. *The C language interface to SQLite Version 2* [online]. Modified 2011-09-21.
http://www.sqlite.org/c_interface.html.
9. *C/C++ Interface For SQLite Version 3* [online]. Modified 2011-09-21.
http://www.sqlite.org/capi3ref.html.
10. STROUSTRUP, Bjarne. *The C++ Programming Language.* Special Edition. AT&T Labs, Florham Park, New Jersey. United States of America: Addison-Wesley, 2000. 1020 p. ISBN 0-201-70073-5.
11. ISO/IEC 14882. *INTERNATIONAL STANDARD: Programming languages — C++.* 11 West 42nd Street, New York, New York 10036: American National Standards Institute, First edition, 1998-09-01. 748 p.
12. Free Software Foundation, Inc. *GCC, the GNU Compiler Collection* [program]. Version 4.4.3, Copyright 2009 Free Software Foundation, Inc. http://www.gnu.org/software/gcc/
13. GCC Bugzilla: Bug 2316 http://gcc.gnu.org/bugzilla/show_bug.cgi?id=2316
14. SUTTER, Herb; ALEXANDRESCU, Andrei. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* 1st edition. Addison-Wesley Professional, 2004. 240 p. ISBN 0321113586.
15. GNU Gama Git source code repository: gama-local directory. Files: sqlitereader.h and sqlitereader.cpp. http://git.savannah.gnu.org/cgit/gama.git/tree/lib/gnu_gama/local/