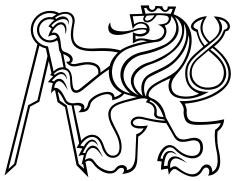
České vysoké učení technické v Praze
Fakulta stavební

# Bakalářská práce

Praha 2011 Václav Petráš

České vysoké učení technické v Praze
Fakulta stavební
Obor geoinformatika



Bakalářská práce

# Podpora databáze SQLite pro program gama-local

## Support of SQLite database in program gama-local

Vedoucí práce: prof. Ing. Aleš Čepek, CSc.
Katedra mapování a kartografie

Praha 2011

Václav Petráš

**ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE**
**Fakulta stavební**
Thákurova 7, 166 29 Praha 6

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

studijní program: Geodézie a kartografie

studijní obor: Geoinformatika

akademický rok: 2010/2011

Jméno a příjmení studenta: Václav Petráš

Zadávající katedra: Katedra mapování a kartografie

Vedoucí bakalářské práce: prof. Ing. Aleš Čepek, CSc.

Název bakalářské práce: Podpora databáze SQLite pro program gama-local

Název bakalářské práce v anglickém jazyce: Support of SQLite database in program gama-local

Rámcový obsah bakalářské práce:

Navrhněte a implementujte podporu databáze SQLite pro program gama-local pro vyrovnání geodetických sítí projektu GNU Gama. Pro komunikaci s databází použijte nativní C/C++ rozhraní databáze. Pro implementaci použijte techniku callback funkcí.

Datum zadání bakalářské práce: _____ Termín odevzdání: **13. 5. 2011**
(vyplňte poslední den výuky příslušného semestru)

Pokud student neodevzdal bakalářskou práci v určeném termínu, tuto skutečnost předem písemně zdůvodnil a omluva byla děkanem uznána, stanoví děkan studentovi náhradní termín odevzdání bakalářské práce. Pokud se však student řádně neomluvil nebo omluva nebyla děkanem uznána, může si student zapsat bakalářskou práci podruhé. Studentovi, který při opakovaném zápisu bakalářskou práci neodevzdal v určeném termínu a tuto skutečnost řádně neomluvil nebo omluva nebyla děkanem uznána, se ukončuje studium podle § 56 zákona o VŠ č. 111/1998. (SZŘ ČVUT čl. 21, odst. 4)

*Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.*

..... .....
vedoucí bakalářské práce vedoucí katedry

**ZDE VLOŽIT ORIGINÁLNÍ ZADÁNÍ**

Zadání bakalářské práce převzal dne: _____

.....
student

Formulář nutno vyhotovit ve 3 výtiscích – 1x katedra, 1x student, 1x studijní odd. (zašle katedra)

Nejpozději do konce 2. týdne výuky v semestru odešle katedra 1 kopii zadání BP na studijní oddělení a provede zápis údajů týkajících se BP do databáze KOS.
BP zadává katedra nejpozději 1. týden semestru, v němž má student BP zapsanou.
(Směrnice děkana pro realizaci studijních programů a SZZ na FSv ČVUT čl. 5, odst. 7)

**Abstrakt**

Program gama-local je součástí projektu GNU Gama a umožňuje vyrovnání lokálních geodetických sítí. Cílem této bakalářské práce je návrh a implementace podpory souborové databáze SQLite v programu gama-local. V současné době program gama-local podporuje jediný vstupní formát, a to XML. Díky této práci získá možnost číst vstupní data i z databáze SQLite. Práce se zabývá specifiky použití callback funkcí v C++ při užití nativního rozhraní databáze SQLite, které je označováno jako SQLite C/C++ API. Součástí práce jsou i testy nově vytvořené verze programu gama-local.

**Klíčová slova:** GNU Gama, vyrovnání geodetických sítí, programování, C, C++, databáze, SQLite

**Abstract**

The program gama-local is a part of GNU Gama project and allows adjustment of local geodetic networks. The aim of this bachelor thesis is to design and implement support for the SQLite database in the program gama-local. Currently, the program gama-local supports only XML as an input. Thanks to this thesis program can read input data from the SQLite database. The thesis deals with the specifics of the use of callback functions in C++ using the native SQLite C/C++ Application Programming Interface. Tests of newly developed version of gama-local are also part of this thesis.

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

Prohlášení

Prohlašuji, že bakalářskou práci na téma „Podpora databáze SQLite pro program gama-local“ jsem vypracoval samostatně. Použitou literaturu a podkladové materiály uvádím v seznamu zdrojů.

V Praze dne .....  .....

(podpis autora)

Poděkování

Děkuji své rodině a blízkým, jenž mi jsou oporou po celou dobu studia. Na tomto místě však především děkuji vedoucímu mé bakalářské práce prof. Ing. Alešovi Čepkovi, CSc. za pomoc, inspiraci a hlavně za spolupráci, bez níž by nebylo možné plnohodnotné dokončení mé práce.

# Obsah

# Úvod

Velká část geodézie se zabývá zpracováním naměřených hodnot, jako jsou například délky a úhly. Výsledkem jsou většinou souřadnice bodů na povrchu Země. Zpracování je tvořeno výpočty, které dnes všechny můžeme zahrnout pod pojem vyrovnání lokální geodetické sítě. Lokální sítí se míní síť vyrovnávaná v kartézské soustavě souřadnic.

Vyrovnání geodetických sítí umožňuje program *gama-local*, který je součástí projektu *GNU Gama*. Zdrojové kódy projektu *GNU Gama* jsou volně dostupné pod licencí *GNU GPL*. Vstupem i výstupem programu *gama-local* jsou v současné době soubory ve značkovacím jazyce XML (*Extensible Markup Language*). Vstupní soubor obsahuje hodnoty měřených veličin a výstupní soubor obsahuje vypočtené (tj. vyrovnané) hodnoty neznámých veličin. Program se ovládá z příkazové řádky.

Cílem této bakalářské práce je umožnit programu *gama-local* číst data z databáze SQLite verze 3. Databáze SQLite je souborová databáze, což znamená, že data nejsou spravována databázovým serverem, ale jsou uložena v jednom souboru. Formát souboru je nezávislý na počítačové platformě, což z databáze SQLite činí vhodný nástroj pro správu a přenos strukturovaných dat mezi různými systémy a programy (nejen v oboru geodézie). K tomuto souboru je možné přistoupit pomocí SQLite C/C++ API, což je rozhraní, které umožňuje programovat aplikace využívající databázi SQLite. Program *gama-local* je napsán v jazyce C++, a tak je možné toto rozhraní, určené pro jazyky C a C++, použít. Tvůrci databáze SQLite C/C++ API poskytují SQLite volně k libovolnému použití.

Podpora databáze v programu *gama-local* bude velmi výhodná z hlediska spolupráce s připravovaným programem *QGama*, který nabízí stejnou funkcionalitu jako program *gama-local*. Program *QGama* není pro příkazovou řádku, ale poskytuje grafické uživatelské rozhraní. V současné době je program *QGama* ve stádiu vývoje.

V programu *gama-local* bude spolupráci s databází SQLite zajišťovat nová třída `SqliteReader`. Tato práce se zabývá implementací této třídy a také funkcemi, které třída používá. Při implementaci je třeba dbát na omezení, která jsou způsobena tím, že SQLite C/C++ API je primárně určeno pro jazyk C, zatímco program *gama-local* je v jazyce C++.

Novou funkcionalitu je nutné po začlenění do programu *gama-local* náležitě otestovat. Projekt *GNU Gama* obsahuje množství příkladů geodetických sítí, které je možné použít jako vstupní data při testování. Vývoj projektu *GNU Gama* probíhá na systémech z rodiny *GNU/Linux*. K testování je tedy možné použít řadu volně dostupných programů, takzvaných unixových utilit. Dále je pro testování možné použít nový program *gama-local-cmp*. Ten doplňuje projekt *GNU Gama* o možnost porovnat výstupy z programu *gama-local*. Program *gama-local-cmp* vyloučí náhodné rozdíly výsledků vznikající při výpočtu na počítači (numerický šum) a je vhodný pro testování nových verzí programu *gama-local*.

# 1 Program *gama-local*

Program *gama-local* je součástí projektu *GNU Gama* [14], který je zaměřen na vyrovnání geodetických sítí. Projekt *GNU Gama* je šířen pod licencí *GNU GPL* [7]. Program *gama-local* je programem pro příkazovou řádku. Je s ním možné vyrovnávat geodetické sítě, které obsahují různé typy měření jako například délky, úhly či výškové rozdíly. V této souvislosti se hodí zmínit, že se v současné době pro projekt *GNU Gama* také vyvíjí multiplatformní grafické uživatelské rozhraní *QGama* [15]. Projekt *QGama* je vyvíjen na platformě *Qt* [13].

Zdrojové kódy projektu *GNU Gama* lze získat stejně jako dokumentaci prostřednictvím internetové stránky projektu:

```
http://www.gnu.org/software/gama/
```

Samotné zdrojové kódy jsou dostupné přímo ze stránky:

```
http://ftpmirror.gnu.org/gama
```

Projekt *GNU Gama* používá pro správu verzí systém Git[1]. Nejlepší možností, jak získat aktuální verzi, je proto příkaz, který vytvoří lokální kopii Git repozitáře.[2]

```
$ git clone git://git.sv.gnu.org/gama.git
```

Aktuální verze programu *gama-local*, kterou je možné získat z výše uvedených míst, již obsahuje i zdrojové kódy, které jsou výsledkem této bakalářské práce.

Na systémech *GNU/Linux* vypadá kompilace a případně i instalace projektu následovně:

```
$ cd gama
$ ./autogen.sh
$ ./configure
$ make
$ make install
```

[1] Oficiální strány systému Git jsou na `http://git-scm.com/`

[2] Pro získání zdrojových kódů z Git repozitáře lze též použít odkaz `http://git.savannah.gnu.org/cgit/gama.git/snapshot/gama-master.tar.gz`.

Po vykonání příkazu `make` by se v adresáři `gama/bin` mělo objevit několik spustitelných souborů, mezi nimi i program *gama-local*.

## 1.1 Podporované formáty

Vstupem pro program *gama-local* bylo doposud jen XML [8]. Jak soubor vypadá, je popsáno v dokumentaci projektu *GNU Gama* [9, str. 7 až 20]. Souboru se obvykle dává přípona `.gkf` (vznikla ze slov „gama konfigurace“).

Výstupem je buď prostý formátovaný text, nebo opět XML. Soubor ve formátu prostého textu je vhodný pro rychlé zjištění výsledků. XML soubor je možné s výhodou použít pro další zpracování.[3] Je například možné výsledky převést do jiného formátu.

## 1.2 Používání

Používání *gama-local* vyžaduje od uživatele základní znalost práce s příkazovou řádkou a dále základní znalost XML. Obecně se předpokládá, že vstupní XML soubor uživatel vytvoří nějakým programem. Je však také možné vyjít z některého z příkladů dostupných ze stránek projektu *GNU Gama* [14] a při malém objemu dat hodnoty do souboru zapsat ručně. Na mém počítači s operačním systémem Ubuntu[4] může vypadat spuštění programu *gama-local* následovně.

```
$ ./gama-local input.gkf --language cz --txt output.txt
```

Předpokládám, že jsem právě v adresáři, kde mám příslušný spustitelný soubor, a že mám vytvořený soubor `input.gkf`, kde jsou příslušné vstupní hodnoty. Kromě jazyka lze z příkazové řádky nastavit například také algoritmus či kódování souborů. Další informace k používání lze nalézt v nápovědě programu či v dokumentaci [9].

[3] Pro čtení výsledků z XML je v projektu *GNU Gama* připravena třída, jejíž popis lze nalézt na `http://geo.fsv.cvut.cz/gwiki/GNU_Gama_LocalNetworkAdjustmentResults`.

[4] Ubuntu je jedna z distribucí operačního systému GNU/Linux. Distribuce je podporována společností Canonical Ltd. a je dostupná z `http://www.ubuntu.com/`

## 1.3 Rozšíření o podporu SQLite databáze

Mnoho aplikací dnes nějakým způsobem využívá databázi SQLite, což je souborová databáze, která používá jazyk SQL. Práce s SQL databází má mnohé výhody, mezi které přirozeně patří pohodlné formulování dotazů. Jak již bylo řečeno, jedná se o souborovou databázi, což znamená, že není potřeba, aby někde běžel databázový server. Rozhraní mezi souborem a daty uloženými v databázi obvykle zajišťuje knihovna. K SQLite autoři dodávají SQLite C/C++ API (viz část 2). Mnoho tvůrců knihoven pro různé programovací jazyky však poskytuje vlastní rozhraní k databázi SQLite. Široká podpora a dostupnost činí z databáze SQLite ideální výměnný formát.

Podpora SQLite databáze by měla usnadnit spolupráci mezi *gama-local* a dalšími programy.

Projekt *QGama* počítá s podporou různých databází díky platformě *Qt*. Mezi nimi je i databáze SQLite. Pro zvýšení vzájemné kompatibility je vhodné, aby i program *gama-local* podporoval databázi SQLite. V projektu *GNU Gama* se na rozdíl od projektu *QGama Qt* nevyužívá. Pro komunikaci s databází SQLite program *gama-local* využívá nativní SQLite C/C++ API.

Databázové schéma, které používá *QGama*, lze použít i pro *gama-local*, čímž je další předpoklad vzájemné kompatibility splněn.

## 1.4 Databázové schéma

Pro projekt *QGama* již bylo vyvinuto databázové schéma. Toto schéma obsahuje popis databázových tabulek, do kterých je možné uložit záznam lokální geodetické sítě. Tento záznam odpovídá jednomu vstupnímu XML souboru a používá se pro něj označení konfigurace (*configuration*). Kromě informací, které lze uložit ve vstupním XML souboru, obsahuje záznam také některé parametry příkazového řádku, se kterými se spouští program *gama-local*.

Databázové schéma bylo vyvinuto s ohledem na to, že *QGama* by měla umožnit propojení s různými SQL databázemi. V databázovém schématu byly použity jen takové konstrukce, které fungují ve všech podporovaných databázích. Schéma je použitelné nebo alespoň částečně použitelné pro tyto databáze: PostreSQL, MySQL,

Oracle Database, SQLite.

Databázové schéma bylo již začleněno do projektu *GNU Gama* a je v souboru `xml/gama-local-schema.sql` v adresáři zdrojových kódů *GNU Gama*. Kompletní databázové schéma je v příloze A.

Pro převod vstupního XML souboru do SQL příkazů (`INSERT`) slouží program *gama-local-xml2sql*. Program je součástí projektu *GNU Gama*. Program generuje SQL příkazy, tyto příkazy by (teoreticky) měly být nezávislé na konkrétní databázi. Pouze pro účely testování databázového schématu byl též napsán program, který údaje (jednu konfiguraci) z SQLite databáze převede do vstupního XML souboru. Používá se pro databázi vytvořenou pomocí zmiňovaného schématu a naplněnou dávkou z programu *gama-local-xml2sql*.

# 2 SQLite C/C++ API

Databáze SQLite verze 3 a vše, co k ní patří, je šířeno pod *Public Domain*. To platí i pro nativní SQLite C/C++ API.

Projekt *GNU Gama* se snaží být co nejméně závislý na jiných knihovnách. V současnosti je závislý pouze na knihovně *Expat* a ta je dodávána spolu s ostatními zdrojovými kódy.[5] Vzhledem k této snaze bylo nativní SQLite C/C++ API jasnou volbou, protože s sebou nepřináší závislost na žádné rozsáhlé knihovně.[6]

SQLite C/C++ API [11] je aplikační programátorské rozhraní pro jazyk C a je uzpůsobeno k tomu, aby mohlo být použito i v jazyce C++. Nicméně na práci v jazyce C++ má toto jisté dopady a to především při použití rozhraní popsaného v části 2.2. Tyto dopady jsou zevrubně rozebrány v části 3.

## 2.1 Klasické rozhraní

Rozhraní tvoří funkce `sqlite3_open`, která přebírá ukazatel na ukazatel na objekt `sqlite3`. Funkce otevře databázi a uloží do proměnné na níž ji byl předán ukaza-

[5] *Expat* je knihovna pro čtení XML. Projekt *GNU Gama* je standardně sestavován se systémovou knihovnou *Expat*. V případě, že je knihovna nedostupná, je možné pro sestavení použít dodané zdrojové kódy.

[6] Navíc je možné sestavit projekt *GNU Gama* bez podpory SQLite databáze (více viz 6.3).

tel hodnotu ukazatele na nový databázový objekt typu `sqlite3`. Jedná se vlastně o konstruktor objektu `sqlite3`, což je objekt reprezentující spojení s databází.

Páteří tohoto rozhraní jsou funkce `sqlite3_prepare_v2` a `sqlite3_step`. Funkci `sqlite3_prepare_v2` se předává ukazatel na databázový objekt, SQL příkaz, který se má vykonat, ukazatel na ukazatel na objekt typu `sqlite3_stmt` a ještě další parametry, které však nejsou důležité (viz [11]). Její zavolání způsobí vytvoření objektu typu `sqlite3_stmt` a lze o ní uvažovat jako o konstruktoru. Objekt `sqlite3_stmt` reprezentuje jeden SQL příkaz.

Funkce `sqlite3_step` pracuje jen s objektem `sqlite3_stmt` a je nutné ji zavolat, aby se příkaz vykonal, neboť funkce `sqlite3_prepare_v2` jej pouze připraví. Pokud byl vykonaný příkaz `SELECT`, tak každé zavolání funkce `sqlite3_step` nás posune na další řádek výsledku příkazu. Jednotlivé hodnoty atributů se pak získávají voláním funkcí ze skupiny `sqlite3_column...` v závislosti na tom, jaký typ hodnoty chceme obdržet. V příkladu, který bude následovat jsou použity funkce `sqlite3_column_int` a `sqlite3_column_text`. To jestli ještě zbývají nějaké nezpracované řádky se zjišťuje z návratového kódu. Pokud vykonaný příkaz nevrací žádný výsledek, stačí volat funkci `sqlite3_step` jen jednou.

Po volání téměř každé z výše jmenovaných funkcí je nutné provádět kontrolu návratového kódu. Jen tak je možné určit, že nastala chyba. Poslední chybovou zprávu lze získat zavoláním funkce `sqlite3_errmsg`, která přebírá ukazatel na objekt `sqlite3`.

Po zpracování výsledků SQL příkazu je třeba uvolnit paměť přidělenou objektu `sqlite3_stmt`. To zajistí funkce `sqlite3_finalize` (je to vlastně destruktor objektu `sqlite3_stmt`). Kontrolu návratového kódu funkce `sqlite3_finalize` není nutné provádět. Ve většině případů zcela postačuje zkontrolovat návratový kód funkce `sqlite3_step`.

Po ukončení práce je třeba zavolat funkci `sqlite3_close`. Kontrola návratového kódu funkce `sqlite3_close` může upozornit mimo jiné na aktivní objekty typu `sqlite3_stmt` (což znamená, že nebylo zavoláno `sqlite3_finalize`).

Následuje příklad, který ukazuje použití výše uvedených funkcí. Pro zjednodušení příklad předpokládá již vytvořenou a naplněnou databázi.

```
1 sqlite3* db;
```

```
2 sqlite3_stmt* stmt;
3 int rc = sqlite3_open("test.sqlite", &db);
4 if (rc)
5   std::cerr << "database error\n";
6 else {
7     rc = sqlite3_prepare_v2(db,
8                             "SELECT id, name FROM people",
9                             -1, &stmt, 0);
10    if (rc)
11        std::cerr <<  "error: "
12                  << sqlite3_errmsg(db) << std::endl;
13    else {
14        while ((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
15            const int id = sqlite3_column_int(stmt, 0);
16            const char* name =
17                 (const char*) sqlite3_column_text(stmt, 1);
18            std::cout << id << " " << name << std::endl;
19          }
20        if (rc != SQLITE_DONE)
21            std::cerr << "error: "
22                      << sqlite3_errmsg(db) << std::endl;
23        sqlite3_finalize(stmt);
24      }
25    sqlite3_close(db);
26  }
```

Použití probíraného rozhraní vyžaduje hodně práce a kódu, vzhledem k tomu, že je stále potřeba kontrolovat návratové kódy, zpracovávat případné chyby a zajišťovat volání funkcí, které plní úlohu konstruktorů a destruktorů. Možným řešením je zabalit tyto funkce do několika objektů, a tím automatizovat určité akce. Další možností je použití rozhraní popsaného v následující části.

Zbývá dodat, že zde bylo popsáno pouze základní rozhraní. SQLite C/C++ API nabízí velké množství funkcí, jejichž popis přesahuje rámec této práce. Příkladem může být skupina funkcí `sqlite3_bind...`, které se využijí především ve spojitosti s příkazem `INSERT`.

## 2.2 Rozhraní s callback funkcemi

SQLite C/C++ API poskytuje ještě další rozhraní, které je vlastně pouhou obálkou nad rozhraním popsaným v části 2.1. Ostatně to praví i dokumentace [10, část 2.0].

> *It is important to realize that neither* `sqlite3_exec` *nor* `sqlite3_get_table` *do anything that cannot be accomplished using the core routines. In fact, these wrappers are implemented purely in terms of the core routines.*

Zmiňovaná funkce `sqlite3_get_table` funguje podobně jako `sqlite3_exec` avšak je označena jako *legacy interface* a je doporučeno ji nepoužívat [11]. To je důvod, proč zde není toto rozhraní rozebráno.

Funkce `sqlite3_exec` je rozhraním, které umožňuje vykonat SQL příkaz jedním jediným voláním. Po zavolání `sqlite3_exec` je nutné zkontrolovat návratový kód a případně uvolnit paměť přidělenou pro chybovou zprávu. Při používání jsme osvobozeni od provádění mnohých kontrol návratových kódů (kontroluje se jen jeden). Před zavoláním funkce `sqlite3_exec` je samozřejmě nutné provést již popsané otevření databáze a po ukončení práce je nutné databázi zase uzavřít. Odpadá však povinnost volat pro každý příkaz trojici funkcí `sqlite3_prepare_v2`, `sqlite3_step` a `sqlite3_finalize`.

V tomto rozhraní se používají callback funkce čili zpětná volání. Již zmiňované funkci `sqlite3_exec` se předá ukazatel na funkci, kterou bude následně volat pro každý řádek výsledku dotazu do databáze. Tato callback funkce se zpravidla píše pro zpracovaní jednoho příkazu. Pokud bude SQL příkaz předaný funkci `sqlite3_exec` příkazem, který nevrací žádný výsledek, nebude callback funkce zavolána. V tomto případě je vhodné předat jako hodnotu ukazatele na callback funkci nulový ukazatel (`NULL`, `0`). Pro vykonávání příkazů, jako jsou například `INSERT` či `UPDATE`, není třeba psát žádnou callback funkci.

Vzhledem k tomu, že kód callback funkce je jinde než kód, který zavolal funkci `sqlite3_exec`, a volající kód zřejmě očekává nějaké výsledky, je nutné předat callback funkci objekt, který bude moci pro uložení výsledků použít. Tento objekt se předá funkci `sqlite3_exec`, která ho následně předá callback funkci. Pro předávání

objektu je použit ukazatel na `void`, je tedy možné předat ukazatel na libovolný objekt. Aby mohl být objekt použit, je nutné `void*` přetypovat na příslušný ukazatel (viz též část 3.4).

Funkce `sqlite3_exec` přebírá pět parametrů. Všechny parametry jsou ukazatele. První dva je nutné zadávat vždy. Jako třetí, čtvrtý a pátý parametr lze předat nulový ukazatel (`NULL`). Parametry jsou uvedeny v následujícím seznamu.

1. `sqlite3*` ukazatel na databázi
2. `const char*` C řetězec s SQL příkazem
3. `int (*)(void*, int, char**, char**)` ukazatel na callback funkci
4. `void*` ukazatel předávaný callback funkci
5. `char**` ukazatel na chybovou zprávu

Jak již bylo naznačeno, callback funkce obsahuje kód, který zpracovává jednotlivé řádky výsledku SQL příkazu. Callback funkce zná obsah vždy jen jednoho řádku. Následuje seznam parametrů callback funkce.

1. `void*` ukazatel předaný funkci `sqlite3_exec`
2. `int` počet atributů
3. `char**` ukazatel na pole atributů
4. `char**` ukazatel na pole jmen atributů

Následuje výpis, který ukazuje základní použití rozhraní tvořeného především funkcí `sqlite3_exec`. Nejprve je otevřena databáze (kontrola návratového kódu je pro zjednodušení vynechána) a inicializován C řetězec pro chybovou zprávu. Poté zavolána funkce `sqlite3_exec`. Funkci je předán jako první parametr ukazatel na databázový objekt typu `sqlite3`, a jako druhý parametr SQL příkaz, který chceme vykonat. Jako třetí je předán ukazatel na callback funkci[7], kterou chceme použít ke zpracování výsledku. Jako čtvrtý parametr je předán ukazatel na objekt, se kterým bude callback funkce pracovat. V tomto případě je to standardní výstupní proud. A jako pátý parametr je předán ukazatel na řetězec s chybovou zprávou.

```
1 extern "C" int readPeople(void* data, int argc,
2                            char** argv, char**);
3 void test() {
```

[7]Ukazatel na funkci v C++ získáme tak, že napíšeme jméno dané funkce.

```
4   sqlite3* db;
5   int rc = sqlite3_open("test.sqlite", &db);
6   // ... check rc
7   char * errorMsg = 0;
8   rc = sqlite3_exec(db, "SELECT id, name FROM people",
9                       readPeople, &std::cout, &errorMsg);
10  if (rc != SQLITE_OK) {
11      std::cerr << "error: ";
12      if (errorMsg) {
13          std::cerr << errorMsg;
14          sqlite3_free(errorMsg);
15        }
16      std::cerr << std::endl;
17    }
18  sqlite3_close(db);
19 }
```

Po zavolání funkce `sqlite3_exec` se provede kontrola návratového kódu (řádek 10) a pokud signalizuje chybu pokračuje se výpisem chybové zprávy. Zde je důležité, že funkce `sqlite3_exec` alokuje paměť pro chybovou zprávu jen v případě, že došlo k chybě. Hodnotu ukazatele na vytvořenou chybovou zprávu pak uloží do proměnné definované na řádku 7. To může provést díky tomu, že jí byl předán ukazatel na tuto proměnou jako pátý parametr. Alokovanou paměť je třeba uvolnit (řádek 14). Nakonec je zavře databáze funkcí `sqlite3_close`.

V předcházejícím výpisu byla callback funkce pouze deklarována. Vzhledem k tomu, že je callback funkci potřeba napsat vždy, když chceme získat výsledek SQL příkazu, uvedu zde i její definici. Na prvním řádku funkce (řádek 3 výpisu) je přetypování ukazatele na `void` na ukazatel na `std::ostream` pomocí operátoru `static_cast`. Situace je zde mírně zesložitěna tím, že přetypovaný ukazatel je použit ještě k získání reference na objekt typu `std::ostream`. Poté je však možno s objektem pracovat, jak je pro něj běžné. V případě výstupního proudu je to vložení dat získaných z databáze. Třetí parametr je ukazatel pole C řetězců s atributy, proto je použit operátor indexování pro přístup k jednotlivým atributům. V příkazu `SELECT` v předcházejícím výpise se vybíraly dva atributy, proto je jasné, že lze použít indexy 0 a 1. Počet atributů uložený ve druhém parametru nebyl využit stejně jako čtvrtý

parametr.

```
1 extern "C" int readPeople(void* data, int argc,
2                            char** argv, char**) {
3   std::ostream& out = *static_cast<std::ostream*>(data);
4   out << argv[0] << " " << argv[1] << std::endl;
5   return 0;
6 }
```

# 3 Propojení C a C++

Tato část uvádí, co je třeba dodržet při propojování kódu napsaného v C a v C++. Těchto zásad se musí držet i implementace třídy `SqliteReader`, protože pracuje s SQLite C/C++ API. V této části předpokládám základní znalosti programovacího jazyka C++, které lze získat například z [5].

## 3.1 Funkce

Jak již bylo naznačeno, při současném používání jazyků C a C++ existují jistá pravidla, která je nutné respektovat. A to se týká i funkcí. Funkce v jazyce C mají jiné linkovací konvence (*linkage conventions*)[8] než funkce v jazyce C++.

Pokud chceme volat C funkci v C++ kódu je nutné specifikovat, že funkce má C linkování (*C linkage*). Specifikace linkování lze dosáhnout tím, že se před deklaraci funkce přidá klíčové slovo `extern` následované řetězcovým literálem označujícím jazyk, v případě jazyka C tedy `extern "C"`. Poznamenejme, že dle standardu [2, 7.5.3] by každá implementace C++ měla poskytnout linkování pro C funkce (`"C"`) a pro C++ funkce (`"C++"`), které je implicitní. Podpora ostatních jazyků je závislá na implementaci. Stejně jako řetězcové literály, jenž je označují. Linkování je možné určit pro jednu funkci.

```
extern "C" int fun(int);
```

Nebo je také možné vložit deklaraci do bloku `extern "C"`.

[8] V české literatuře se často pracuje s pojmem zacházení se jmény funkcí.

```
extern "C" {
  int fun(int);
}
```

Aby mohl být jeden hlavičkový soubor použit pro C i C++, je nutné, aby funkce měly C linkovnání. Toho se dosáhne pomocí bloku `extern "C"` a direktiv preprocesoru.

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

Tento postup často používají knihovny pro C pro své hlavičkové soubory, aby mohly být použity s C++. Uvedené platí to i pro SQLite C/C++ API (hlavičkový soubor `<sqlite3.h>`).

## 3.2 Ukazatele na funkce

Různé druhy linkování se týkají i ukazatelů na funkce a to jak typů deklarovaných pomocí `typedef`, tak i parametrů, které jsou ukazateli na funkce. Jak se určí linkování je ukázáno na příkladech.

```
1 // mixing C and C++ function pointers:
2 typedef void(*pf_cpp)(); // pf_cpp is a pointer to a C++
                           function (has C++ linkage)
3 void f_cpp() {} // f_cpp is a C++ function (has C++ linkage)
4 extern "C" {
5   typedef void(*pf_c)(); // pf_c is a pointer to a C
                             function (has C linkage)
6   void f_c() {} // f_c is a C function (has C linkage)
7 } // end of extern "C" block
8
9 void fun1(pf_cpp pf) { // fun1 takes one parameter of type
                           pointer to C++ function
```

```
10   pf();
11 }
12
13 void fun2(pf_c pf) { // fun2 takes one parameter of type
                            pointer to C function
14   pf();
15 }
16
17 void test_fun12() {
18   fun1(f_cpp); // ok
19   fun2(f_c);   // ok
20   fun1(f_c);   // error
21   fun2(f_cpp); // error
22 }
```

Když předáme funkci, očekávající ukazatel na C++ funkci, ukazatel na C funkci (řádek 20), nebo naopak předáme místo ukazatele na C funkci ukazatel na C++ funkci (řádek 21), mělo by dojít k chybě při kompilaci. To odpovídá tomu, co je uvedeno ve standardu [2, 7.1.5]:

> *Two function types with different language linkages are distinct types even if they are otherwise identical.*

Na druhou stranu B. Stroustrup uvádí, že pokud to implementace umožňuje, je možné, aby ukazatel na C funkci stál na místě ukazatele na C++ funkci a obráceně, tedy aby tyto konverze byly rozšířením jazyka [1, str. 208].

Kompilátor *GCC*[9] výše uvedené opravdu umožňuje. Je tedy možné ukazatele libovolně zaměňovat. Dokonce lze do proměnné typu ukazatel na C funkci přiřadit ukazatel na statickou metodu třídy. Vzhledem k tomu, že toto chování není přenositelné, bylo by dobré, aby *GCC* poskytlo při kompilaci nějaké varování. Bohužel tomu tak není, protože *GCC* pro typy různá linkování nerozlišuje.[10] Ze stejného důvodu není

[9] Název *GCC* (*GNU Compilers Collection*) označuje celou sadu kompilátorů. Kompilátor pro C++ z této sady se jmenuje *g++*. Nadále však budu používat souhrnný název *GCC*, neboť je to běžná praxe.

[10] To, že *GCC* nerozlišuje typy s různým linkováním, je považováno za chybu, viz GCC Bugzilla – Bug 2316 `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=2316`

například možné přetěžovat funkce na základě typů, které se liší pouze v linkování.

## 3.3 Viditelnost funkcí

Definice funkcí mají v C i C++ globální viditelnost. Deklarace funkce je však viditelná jen v dané překladové jednotce (ve zdrojovém souboru s vloženými hlavičkovými soubory). Funkci proto můžeme použít pouze v ní. Pokud chceme použít funkci v jiné překladové jednotce, než ve které je funkce definována, musíme do zdrojového kódu vložit její deklaraci. To se zpravidla učiní vložením příslušného hlavičkového souboru. V následujících příkladech je však použita možnost přímého napsání deklarace funkce, tj. bez použití hlavičkového souboru.

Někdy je však výhodné znemožnit použití funkce mimo překladovou jednotku, kde je definována, například kvůli skrytí implementace či prostému zabránění konfliktu jmen. Skrytí funkce se v jazyce C++ provede tak, že se funkce umístí do bezejmenného (nepojmenovaného, anonymního) prostoru jmen, jak je vidět na řádku 6 následujícího výpisu (výpis představuje dva soubory). Funkci pak není možné použít, ani když poskytneme příslušnou deklaraci. Překladač ji totiž považuje za jinou funkci. Různé možnosti přístupu k funkcím jsou ukázány na následujícím výpisu. Na řádku 2 je definice funkce a v jiném souboru je její odpovídající deklarace (ve výpisu řádek 10). Pokud se pokusíme použít funkci, kterou jsme nedeklarovali, překladač ohlásí chybu. Chyba nastane také v případě, že se pokusíme poskytnout deklaraci funkce z bezejmeného prostoru jmen (řádek 12). Deklarovali jsme totiž novou funkci, jejíž definice není známa.

```
1 // a.cpp:
2 int function_k(int a) { return a; }
3 int function_l(int a) { return a; }
4 int function_m(int a) { return a; }
5 namespace {
6   int function_n(int a) { return a; }
7 }
8
9 // b.cpp:
10 int function_k(int);
```

```
11 int function_m(int b) { return b; } // error: multiple
                                          definition of
                                          function_m(int)[11]
12 int function_n(int);
13
14 void test_kln() {
15   function_k(0);
16   function_l(1); // error: function_l was not declared in
                                            scope
17   function_n(2); // error: undefined reference to
                                            function_n(int)
18 }
```

V jazyce C se skrytí funkce provede tak, že se do deklarace funkce doplní klíčové slovo `static`.

```
static void function_a(int) { }
```

Viditelnost definice pak bude omezena pouze na danou překladovou jednotku. Funguje to tedy analogicky jako bezejmený prostor jmen v C++.

Nutno poznamenat, že nemá smysl vkládat funkce v bezejmenný prostoru jmen do hlavičkových souborů. To samé platí i pro funkce deklarované jako `static`.

### 3.3.1 Deklarace `extern "C"` a prostor jmen

Pokud funkci deklarujeme jako `extern "C"` uvnitř nějakého prostoru jmen, musíme se na ni odvolávat pomocí prostoru jmen, i když má C linkování (to je vidět na řádcích 4, 12 a 18 následujícího výpisu). Funkci nelze použít bez specifikace prostoru jmen (řádek 19). Je však možné poskytnout deklaraci bez specifikace prostoru jmen a to i přesto, že funkce byla původně deklarována v prostoru jmen. Funkce se pak používá bez specifikace prostoru jmen. Tento postup ukazují řádky 5, 14 a 20. Tento postup na funkce s C++ linkováním použít nelze. Pokud se o to pokusíme, deklarujeme novou funkci.

```
1 // a.cpp:
2 namespace foo {
```

[11] Chybové hlášky pochází z kompilátoru *GCC*. Stejně je tomu i v následujícím textu.

```
3   extern "C" {
4     int bar() { return 1; }
5     int bar_d(double) { return 1; }
6   }
7   void bar_cpp() {}
8 }
9
10 // b.cpp:
11 namespace foo {
12   extern "C" int bar();
13 }
14 extern "C" int bar_d(double);
15 void bar_cpp(); // not namespace foo{ void bar_cpp(); }
16
17 void test_foo_bars() {
18   foo::bar(); // ok
19   bar();      // error: bar was not declared in this scope
20   bar_d(3);   // ok
21   bar_cpp();  // error: undefined reference to bar_cpp()
22 }
```

Kdybychom v deklaraci na řádku 14 vynechali `extern "C"`, deklarovali bychom novou funkci (s C++ likováním), ke které by chyběla definice.

### 3.3.2 Deklerace `extern "C"` a viditelnost

V C++ se deklarace `extern "C"` často používá pro práci s rozhraním a společnými hlavičkovými soubory pro C i C++ (viz část 3.1). Pokud se deklarace `extern "C"` používá kvůli předávání ukazatelů na funkce nějaké C knihovně, můžeme chtít, aby definice funkce nebyla přístupná mimo danou překladovou jednotku. V C++ se pro omezení přístupu použije bezejmenný prostor jmen (viz část 3.3). Funkce deklarované jako `extern "C"` však mohou být použity i bez specifikace prostoru jmen (viz část 3.3.1). Následující výpis ukazuje, jak `extern "C"` funguje s bezejmenným prostorem jmen.

```
1 // a.cpp:
```

```
2 namespace {
3   extern "C" int bar_i(int) { return 1; }
4 }
5
6 extern "C" double bar_d(double) { return 1; }
7
8 // b.cpp:
9 extern "C" {
10   int bar_i(int);
11   double bar_d(double);
12 }
13
14 void test_bars() {
15   bar_i(1);   // ok?
16   bar_d(1.0); // ok
17 }
```

Na řádku 3 je deklarována a definována funkce uvnitř bezejmenného prostoru jmen. Pokud by se jednalo pouze o C++ nebylo by možné ji použít v jiné překladové jednotce. Avšak funkce se řídí pravidly jazyka C, a proto je bezejmenný prostor jmen ignorován. Na řádku 10 poskytneme deklaraci funkce a na řádku 15 ji použijeme. Kód se sice přeloží, ale vůbec to není to, čeho jsme chtěli dosáhnout, tedy skrytí funkce v překladové jednotce.

Funkce deklarovaná v bezejmenném prostoru jmen jako `extern "C"` se chová stejně, jako kdyby byla deklarována mimo bezejmenný prostor jmen. Vzhledem k tomu, že se jedná o C funkci, nabízí se řešení pomocí klíčového slova `static` (viz část 3.3). To je ukázáno na následujícím výpisu. Při překladu dojde k chybě (řádek 10), protože definice funkce je nedostupná, a to je přesně to, co jsme zamýšleli.

```
1 // a.cpp:
2 extern "C" {
3   static void bar_c(char) { }
4 }
5
6 // b.cpp:
7 extern "C" int bar_c(char);
```

```
8
9  void test_bar_c() {
10   bar_c('a');     // error: undefined reference to bar_c
11 }
```

### 3.3.3 Použití při implementaci třídy `SqliteReader`

Callback funkce používané v SQLite C/C++ API (2.2) je nutné deklarovat jako `extern "C"`, protože SQLite C/C++ API očekává C funkci (viz 3.1).

By bylo vhodné skrýt callback funkce v překladové jednotce (viz výše). Nelze však použít bezejmenný prostor jmen, protože se jedná o C funkce. Pro ně je možné použít deklaraci `static`.

Bohužel se nepodařilo ověřit, zda řešení, které kombinuje deklarace `extern "C"` a `static`, je přenositelné. V *GCC* daná kombinace funguje přesně tak, jak to bylo uvedeno v předcházejících částech. V *GCC* je implementace C a C++ velmi těsně propojena a to je nejspíše důvod, proč deklarace `extern "C"` a `static` fungují dohromady. Nelze však spoléhat na to, že s jinými kompilátory tato kombinace bude také fungovat, a proto bylo při implementaci třídy `SqliteReader` použito řešení bez použití klíčového slova `static`.

Callback funkce, jejichž platnost by bylo dobré omezit jen na danou překladovou jednotku, jsou dostupné v celém programu. Konfliktu jmen je bráněno prefixem `sqlite_db_` v názvu funkcí. Co se týče rizika špatného použití, spoléhá se na disciplínu programátora. Funkce vlastně není možné k ničemu použít, protože pracují s objektem `ReaderData`, který je jim nutné předat. Jeho deklarace však není mimo implementaci třídy `SqliteReader` dostupná.

## 3.4 Předávání objektů

Při práci se SQLite C/C++ API za použití callback funkcí je nutné předávat ukazatel na objekt, se kterým pracujeme a potřebujeme ho v callback funkci. Předání se děje pomocí ukazatele na `void` (viz část 2.2). Ukazatel na `void` může v C a C++ ukazovat na libovolný typ. Konverze libovolného ukazatele na typ `void*` je v obou

jazycích implicitní. Konverze z typu `void*` na jiný ukazatel se v C++ provede pomocí operátoru `static_cast`, v C je konverze implicitní. V C se ukazatel na `void` používá tam, kde je třeba předávat různé typy objektů. Předání parametru callback funkci tak, jak je tomu v SQLite C/C++ API, je typickým příkladem. Následně je nutné přetypovat `void*` zpět na ukazatel na požadovaný typ. V C++ se těmto konstrukcím snažíme vyhnout. Avšak při práci s knihovnou v C se jim vyhnout nemůžeme.

Jako příklad dobře poslouží použití SQLite C/C++ API (viz 2.2). Zavolání funkce `sqlite_exec`, která volá callback funkci vypadá v případě funkce následovně.

```
sqlite3_exec(sqlite3Handle, queryString, readPoints,
                 readerData, &errorMsg);
```

Použitá callback funkce `readPoints` pak může vypadat třeba takto.

```
extern "C" int readPoints(void* data, int argc,
                          char** argv, char**)
  {
    ReaderData* d = static_cast<ReaderData*>(data);
    // ... callback's code
  }
```

Vzhledem k tomu, že funkce pracuje především s objektem `ReaderData`, mohla by se tato práce provést v nějaké metodě třídy (struktury) `ReaderData`. Callback funkce by pak pouze provedla přetypování a zavolala příslušnou metodu. Takové funkci se často říká trampolínová a pokud se budeme řídit tím, co je uvedeno v částech 3.1 a 3.5, stane se C obálkou nad metodou daného objektu.

```
extern "C" int readPoints(void* data, int argc,
                          char** argv, char**)
  {
    ReaderData* d = static_cast<ReaderData*>(data);
    d->readPoints(argc, argv);
  }
```

Poznamenejme, že naše C funkce může zavolat metodu objektu, jen pokud je veřejná, nebo pokud je C funkce deklarovaná jako přítel (`friend`)[12] třídy daného

[12] V jazyce C++ mohou přátelé tříd přistupovat k jejich soukromým členům.

objektu. Při deklaraci přátelské funkce s C linkováním musíme nejprve deklarovat funkci jako `extern "C"` a poté ji teprve deklarovat jako přítele třídy. Obě deklarace nelze spojit do jedné a je nutné je provést právě v tomto pořadí.

```
extern "C" int readPoints(void*, int, char**, char**);
class ReaderData
  {
    // ...
    friend int readPoints(void* , int, char**, char**);
    // ...
  }
```

Při implementaci třídy `SqliteReader` (přesněji při implementaci pomocné struktury `ReaderData`) byla použita první uvedená možnost. Zmiňovaná C funkce se tedy neomezuje na pouhé zavolání metody příslušného objektu, ale vykonává všechny potřebné činnosti. Část 4 pojednává o tom, jak funkce přistupuje k datovým složkám objektu.

## 3.5 Zpracování výjimek

Jazyk C++ poskytuje pro zpracování chybových stavů mechanismus výjimek. Mezi jeho výhody patří lepší oddělení kódu, který ošetřuje chybový stav, od kódu, který zajišťuje normální fungování programu. Další výhodou výjimek oproti jiným způsobům práce s chybovými stavy je to, že nutí programátora je ošetřit (návratový kód lze ignorovat, zatímco nezachycená výjimka způsobí pád programu). Jazyk C však mechanismus výjimek nemá a pro ohlašování a zjišťování chybových stavů používá jiné metody.[13]

Pokud v C++ používáme knihovnu, která je určená pro jazyk C, musíme se s rozdílným přístupem k chybovým stavům vyrovnat. Je totiž možné, že námi napsaná funkce bude volána v rámci C knihovny. A to je právě případ callback funkcí, kdy ukazatel na naši funkci předáme nějaké C funkci, která ji pak pomocí ukazatele zavolá. Pokud by v naší funkci došlo k vyvolání výjimky a ona výjimka by opustila tělo naší funkce, způsobilo by to pád programu z důvodu nezachycené výjimky.

[13] V jazyce C sice existují rozšíření, která umožňují práci s výjimkami, ale tato řešení rozhodně nejsou přenositelná.

Možný výstup programu přeloženém pomocí *GCC* je na následujícím výpisu.

```
terminate called after throwing an instance of
                 'std::runtime_error'
  what():  some error message
```

Z uvedeného vyplývá, že callback funkce musí všechny výjimky, které v ní byly vyvolány, zachytit. Kód v těle callback funkce, který může vyvolat výjimku, musí být obalen blokem `try-catch`. Bloky `catch` musí zachytit každou výjimku vyvolanou v bloku `try`, takže posledním `catch` blokem musí být `catch` blok s výpustkou. Jednoduchá ukázka je na následujícím výpisu.

```
1 int someCallback(int a) {
2   // can not throw exception
3   try {
4     // can throw exception
5   }
6   catch (std::exception& e) {
7     // handle exception
8   }
9   catch (...) {
10    // handle unknown exception
11  }
12 }
```

Stále ovšem zbývá vyřešit, jak nahlásit, že došlo k výjimce. Nahlášení je nutné provést ve stylu jazyka C. Knihovna, se kterou pracujeme, by měla mít definovaný nějaký způsob, jak callback funkce ohlašují, že došlo k chybě. V případě SQLite C/C++ API je to návratový kód (viz 2.2). Toto řešení však neobsahuje způsob, jak zjistit, ke které výjimce došlo. I když v případě návratového kódu lze například přiřadit různým výjimkám různé návratové kódy. To ovšem není moc pohodlné řešení a ještě není zaručeno, že nám knihovní funkce návratový kód callback funkce sdělí.

Zcela jinou možností, jak se zbavit problémů s výjimkami při práci s C knihovnou, je výjimky vůbec nepoužívat. Tím bychom se ovšem připravili o výhody používaní výjimek. A v případě, že používáme kód, který nemůžeme změnit a výjimky vyvolávat může (například standardní knihovnu C++), nemáme jinou možnost než

výjimky zachytávat (alespoň pomocí `catch` s výpustkou).

Jak se výjimky ošetřují v implementaci třídy `SqliteReader` a jak lze zabránit ztrátě informace o tom, jaká výjimka byla vyvolána, pojednává část 5.

## 3.6 Komplexní řešení pomocí šablon a maker

Použití SQLite C/C++ API (viz 2.2) s callback funkcemi vede k tomu, že je třeba napsat několik podobných funkcí. Funkce jsou podobné v tom, že tělo každé funkce musí obsahovat tentýž kód. Jedná se především o kód, který zajistí odchytávání výjimek a který nemusí být triviální (viz část 5). Důvody, proč tomu tak musí být, jsou uvedeny v částech 3.4 a 3.5. Kód callback funkcí lze tedy rozdělit na společný kód a kód, který vykonává operace týkající se přímo účelu dané funkce. Společný kód je tedy možné přesunout do jedné funkce. Vzhledem k tomu, že kód chceme přidat již během kompilace, je ideálním řešením použití šablonové funkce. Jejím parametrem bude ukazatel na konkrétní funkci, kterou chceme obalit kódem, který odchytává všechny výjimky.

Nejprve uveďme dva typy ukazatelů na funkce. Jak bylo uvedeno v části 3.2, jedná se o rozdílné typy.

```
1 extern "C" typedef int (*CCallbackType)
2                             (void*, int, char**, char**);
3 typedef int (*CppCallbackType)
4                             (void*, int, char**, char**);
```

Následuje kód šablonové funkce, která zajistí odchycení výjimek. Kód je značně zjednodušen a ponechávám také stranou, zda je šablonovým parametrem hodnota typu `CCallbackType` či `CppCallbackType`.

```
1 template<CallbackType callback>
2         int Data::saveCallbackCaller(void* data, int argc,
3                                      char** argv,
4                                      char** cnames)
5   {
6     Data* p = static_cast<Data*>(data);
7     try {
8         return callback(p, argc, argv, cnames);
```

```
9       }
10    catch (Exception& e) {
11        // ... store an exception in p
12      }
13    catch (...) {
14        // ... store an exception in p
15      }
16    return 1;
17  }
```

V callback funkci lze bez jakýchkoli okolků vyvolávat výjimky. Díky tomu, že je obalena uvedenou šablonovou funkcí, nedojde k pádu programu. Jak může vypadat základ callback funkce je uvedeno na následujícím výpisu.

```
1 int Data::readPoints(void* data, int argc,
2                       char** argv, char** cnames) {
3     // ... do something
4     // can throw exception
5     return 0;
6   }
```

Pokud funkce `sqlite3_exec` pomocí návratového kódu nahlásí, že došlo k chybě, je nutné zpracovat informace o uložené výjimce. Vzhledem k tomu, bylo by vhodné obalit i funkci `sqlite3_exec` funkcí, které se o zpracování postará. Tato obálka může zajistit například znovu vyvolání výjimky či vyvolání výjimky v případě chyby databáze. Ve výpisu funkce `exec` (obálky funkce `sqlite3_exec`) ponechávám jen kód týkající se předávání ukazatele na callback funkci (kód, který zpracovává výjimky lze najít v části 5.3).

```
1 void Data::exec(sqlite3* sqlite3Handle,
2                 const std::string &query,
3                 CCallbackType callback, void* data)
4 {
5   // ...
6   int rc = sqlite3_exec(sqlite3Handle, query.c_str(),
7                         callback, data, &errorMsg);
8   // ...
9 }
```

Kdybychom zapomněli, že `CCallbackType` a `CppCallbackType` jsou jiné datové typy, mohli bychom funkci `exec` používat následovně.

```
exec(sqlite3Handle, query,
     saveCallbackCaller<readPoints>, this);
```

Dokonce bychom mohli změnit i funkci `exec` na šablonovou. Jako parametr šablony by samozřejmě měla hodnotu ukazatele na callback funkci. Nic z toho však není možné kvůli rozdílnosti typů `CCallbackType` a `CppCallbackType`. Šablonové funkce musí mít nutně C++ linkování, a proto ukazatel na ně nelze použít jako ukazatel na C funkci. Pro použití s C je nutné každou instanci šablonové funkce obalit do funkce, které je deklarována jako `extern "C"`.

```
1 extern "C" {
2     int c_readPoints(void* data, int argc,
3                      char** argv, char** cnames) {
4         return Data::saveCallbackCaller<Data::readPoints>
5                                     (data, argc, argv, cnames);
6       }
7   }
```

Uvedené obalení nelze ze zřejmých důvodů automatizovat pomocí šablon. Lze však použít makro.

```
1 #define IMPLEMENT_C_CALLBACK(FUNCTION_NAME) int c_\
2 ##FUNCTION_NAME\
3 (void* data, int argc, char** argv, char** cnames) { return
                  Data::saveCallbackCaller<Data::\
4 FUNCTION_NAME\
5 >(data, argc, argv, cnames); }
```

Makro vytvoří funkci s C linkováním, která zavolá funkci, jenž je instancí šablony zajišťující, že žádná výjimka neopustí tělo dané funkce. Použití makra je jednoduché (volání makra není ukončeno středníkem vzhledem k tomu, že definice funkce středníkem nekončí). Parametrem je název funkce, kterou potřebujeme obalit.

```
IMPLEMENT_C_CALLBACK(callback_readClusters)
```

Použití funkce, které je výsledkem práce makra je očekávané.

```
exec(sqlite3Handle, query, c_callback_readPoints, this);
```

Nutno říci, že vzhledem k tomu, že bylo stejně použito makro, by bylo možné nepoužít šablonovou funkci vůbec a nahradit ji makrem. To by ale makro velmi prodloužilo. Použití šablony je bezpečnější a obecně doporučované řešení problému tohoto typu.

Výjimku zachycenou v těle callback funkce je vhodné nějakým způsobem uložit a případně ji ve vhodný okamžik zase vyvolat. O tom, jak to provést, pojednává část 5.

Toto řešení nebylo v konečné implementaci třídy `SqliteReader` použito. Důvodem je jeho obtížná pochopitelnost pro čtenáře kódu.

# 4 Soukromá implementace

V C++ je pravidlem, které se většinou dodržuje, že datové členy tříd se deklarují jako soukromé. Tyto soukromé členy jsou sice nedostupné, ale jsou stále viditelné. Pokud je chceme opravdu skrýt můžeme použít *soukromou implementaci*. Soukromá imlementace (*Private Implementation, Pimpl*) bývá někdy zařazována mezi návrhové vzory.

Třída `SqliteReader` má v objektu soukromé implementace všechny datové členy. Avšak obecně lze do soukromé implementace umístit i všechny soukromé metody. Třída `SqliteReader` obsahuje ukazatel na strukturu typu `ReaderData`, který je implementačním objektem. Při deklaraci třídy se soukromou implementací, lze využít dopředné deklarace (*forward declaration*). Použití pak vypadá následovně.

```
1 // sqlitereader.h:
2 struct ReaderData; // forward declaration
3
4 class SqliteReader
5 {
6 public:
7   // ...
8 private:
9   // ...
```

```
10   ReaderData* readerData; // pointer to private data
11 };
```

Díky dopředné deklaraci je možné použít ukazatel na strukturu `ReaderData`, avšak to, jak skutečně struktura vypadá, zůstává skryto. Deklarace struktury je až v implementačním souboru. Přístup k jejím členům je proto možný jen tam. Obecně bývá výhodné deklarovat strukturu v privátní části třídy. Pro `SqliteReader` tomu tak ale není. Je totiž nutné, aby ke struktuře a jejím členům měli přístup callback funkce, které nejsou a nemůžou být jejími metodami. Implementační objekt je deklarován skutečně jako struktura (`struct`) se všemi členy veřejnými. Mimo jiné právě kvůli callback funkcím.

Pokud bychom strukturu `ReaderData` deklarovali jako soukromou a vnořenou ve třídě `SqliteReader` a ve struktuře `ReaderData` zavedli soukromou část, callback funkce bychom museli deklarovat jako přátelské. Přátelství bychom museli deklarovat jak ve třídě `SqliteReader`, kvůli přístupu k soukromé struktuře `ReaderData`, tak i ve struktuře `ReaderData`, kvůli přístupu k jejím soukromým členům. Alternativně bychom mohli vybavit strukturu `ReaderData` sadou přístupových funkcí. To vše je však zbytečná komplikace. Callback funkce vlastně hrají roli metod struktury `ReaderData`, a proto k ní mají přístup. Ze stejného důvodu mají přístup i k jejím soukromým složkám.

Při použití soukromé implementace je nutné vyřešit kopírování a přiřazení, aby se nestalo to, že dva objekty budou sdílet stejný implementační objekt (to obvykle totiž není naším cílem). Třída `SqliteReader` má zakázaný kopírovací konstruktor a operátor přiřazení. Kopírování nedává v jejím případě smysl, protože reprezentuje jeden zdroj (jeden databázový soubor a spojení s ním). Kopírování implementačního objektu je tímto pro třídu `SqliteReader` vyřešeno.

Mezi výhody soukromé implementace patří veliký stupeň oddělení rozhraní od implementace. Díky použití dopředné deklarace není nutné, aby hlavičkový soubor vkládal jiné hlavičkové soubory kvůli tomu, aby mohl mít datové složky daných typů. Tyto hlavičkové soubory se vkládají až do implementačního souboru. Další často uváděnou výhodou je rychlejší kompilace. Tato výhoda se však projeví jen u větších tříd a projektů. Nevýhodou soukromé implementace je především větší složitost kódu.

Použité soukromé implementace se podobá vytvoření obálky kolem jiné třídy. Soukromá impleplementace bývá také označována jako *Pointer to Implementation* či *Private Class Data*. Souvisí také s návrhovým vzorem Most (*Bridge*). Při implementaci třídy `SqliteReader` byla však použita výše popsaná základní podoba soukromé implementace.

# 5 Polymorfní práce s výjimkami

Základem pro polymorfní práci s výjimkami je klonování (*cloning*). Informace o něm lze získat například z [1, str. 424]

## 5.1 Klonování

V případě, že potřebujeme kopírovat objekt, jehož přesný typ neznáme (máme jen referenci či ukazatel), nemůžeme použít kopírovací konstruktor. Ten totiž není a nemůže být virtuální a to nám zabraňuje pracovat polymorfně. Jeho použití, vědomé či nevědomé, může vést k ořezání (dělení) objektu (*slicing*).

Následující výpis ukazuje, jak snadno dojde k ořezání objektu, tj. zavolá se kopírovací konstruktor základní třídy, i když pracujeme s odvozenou. K zavolání kopírovacího konstruktoru dojde, když objekt předáváme hodnotou. Nový objekt bude vždy objektem základní třídy nehledě na to, jakým byl ten původní. Připravíme se tak o polymorfní chování. Jak je také vidět, předáváme-li objekt referencí, ke kopírování nedojde, a tak nedojde ani k ořezání.

```
1 class Base {
2 public:
3     virtual ~Base() { }
4     virtual std::string name() { return "Base"; }
5   };
6 class Derived : public Base {
7 public:
8     virtual std::string name() { return "Derived"; }
9   };
10
```

```
11 void print_val(Base b) {
12     std::cout << "name is " << b.name() << std::endl;
13   }
14 void print_ref(Base& b) {
15     std::cout << "name is " << b.name() << std::endl;
16   }
17
18 void test() {
19     Derived d;
20     print_val(d); // prints: name is Base
21     print_ref(d); // prints: name is Derived
22   }
```

Může se však stát, že objekt musíme korektně zkopírovat, i když nevíme jeho přesný typ. A tak tomu také je, když chceme ukládat výjimky (viz též 3.5). Místo kopírování je v tomto případě nutné použít klonování (*cloning*). Při klonovaní voláme místo kopírovacího konstruktoru virtuální funkci, která volá kopírovací konstruktor nového objektu. Virtuální funkce zná skutečný typ objektu a zavolá tedy správný kopírovací konstruktor. Klonovací funkce zpravidla vytváří nový objekt pomocí `new` a vrací ukazatel na nový objekt (přidělenou paměť uvolňuje volající).

Předchozí příklad je v následujícím výpisu přepsaný tak, aby funkce použitá na řádku 23 (následujícího výpisu) měla vlastní kopii objektu. Díky klonování ji má a zároveň se jedná objekt správného typu (tedy opravdu o kopii původního objektu).

```
1 class Base {
2 public:
3     virtual ~Base() { }
4     virtual Base* clone() { return new Base(*this); }
5     virtual std::string name() { return "Base"; }
6   };
7 class Derived : public Base {
8 public:
9     virtual Derived* clone() { return new Derived(*this); }
10    virtual std::string name() { return "Derived"; }
11  };
12
```

```
13 void print_val(Base* b) {
14     std::cout << "name is " << b->name() << std::endl;
15   }
16 void print_ref(Base& b) {
17     std::cout << "name is " << b.name() << std::endl;
18   }
19
20 void test() {
21     Derived d;
22     Base* b = d.clone();
23     print_val(b); // prints: name is Derived
24     print_ref(d); // prints: name is Derived
25     delete b;
26   }
```

Pokud zároveň v základní třídě označíme kopírovací konstruktor za chráněný, znemožníme tím okolí kopírovat objekty a díky tomu také nemůže dojít k nechtěnému ořezání objektu. Každý potomek základní třídy musí definovat klonovací metodu. Vzhledem k tomu, že základní třída bude nejspíše třídou abstraktní nic nebrání tomu, aby klonovací metoda byla čistě virtuální. Díky tomu každý přímý potomek naší základní třídy bude muset definovat vlastní klonovací metodu.[14]

Obdobným postupem, jaký se použije při klonování, je možné tvořit nové objekty obecně, nikoli jen kopie (klony). Dokonce tak lze řešit i úplně jiné věci, jako je například vyvolávání výjimek (viz následující část).

## 5.2 Ukládání a vyvolávání výjimek

Standardní zpracování výjimek vypadá následovně. Někde uvnitř bloku `try` se vyvolá výjimka, například uvnitř volání nějaké funkce. Samotného vyvolání se dosáhne příkazem `throw`. Výjimka poté putuje tak dlouho než se dostane k příslušnému bloku `catch`, který ji zachytí. Důležité přitom je, že výjimky by se měly vyvolávat hodnotou a zachytávat odkazem.

[14] Jak vidno, není zajištěno, že nepřímý potomek definuje svou vlastní klonovací metodu. Jak zajistit s využitím nevirtuálního rozhraní, že potomci své vlastní klonovací metody definují, je popsáno v [4, typy 39 a 54].

Následuje výpis, kde jsou dvě třídy, které budu používat v dalších příkladech. Budu je používat jako třídy výjimek. Obsahují metody pro klonování a pro snadné zjištění typu.

```
1 class Base {
2 public:
3     virtual ~Base() { }
4     virtual Base* clone() { return new Base(*this); }
5     virtual std::string name() { return "Base"; }
6   };
7 class Derived : public Base {
8 public:
9     virtual Derived* clone() { return new Derived(*this); }
10    virtual std::string name() { return "Derived"; }
11  };
```

Obě uvedené třídy mají veřejný kopírovací konstruktor (doplní jej překladač). Veřejný kopírovací konstruktor je nutný k tomu, aby výjimka mohla být vyvolána pomocí `throw`. Jak vypadá standardní zpracování výjimek, je uvedeno na následujícím jednoduchém výpise. Poznamenejme, že na pořadí bloků `catch` záleží.

```
1 try {
2     throw Derived();
3   }
4 catch (Derived& e) {
5     std::cout << "Derived exception" << std::endl;
6   }
7 catch (Base& e) {
8     std::cout << "Base exception" << std::endl;
9   }
```

Teď ovšem uvažme možnost, uvedenou v části 3.5. Totiž tu, kdy potřebujeme všechny výjimky zachytit a ještě předat informaci o tom, jaká výjimka byla zachycena. Ideálním řešením je vytvořit kopii zachycené výjimky, někam si ji uložit (například do objektu, se kterým právě pracujeme) a posléze (až bude kontrola nad během programu zase v našich rukou) si ji odtud zase vyzvednout. Kopii však nemůžeme vytvořit kopírovacím konstruktorem, protože neznáme přesný typ

výjimky. Ten bychom znali, jen když bychom měli blok `catch` pro každý typ výjimky. To ale není dobré řešení. Musíme znát všechny typy výjimek a navíc vždy, když se do hierarchie výjimek přidá nová, musíme ji přidat i sem. Klonování je řešením našeho problému. Aniž bychom museli zjišťovat přesný typ výjimky získáme její přesnou kopii (viz též část 5.1). Následující výpis demonstruje naklonování výjimky.

```
1 Base* a = 0;
2 try {
3     throw Derived();
4   }
5 catch (Base& e) {
6     a = e.clone();
7   }
8 if (a) {
9     std::cout << a->name() << std::endl; // prints: Derived
10    delete a;
11  }
```

Nicméně, to, že výjimku ukládáme a dokážeme z ní přečíst údaje, ještě nemusí stačit. Je totiž možné, že výjimku budeme muset znovu vyvolat, a to proto, že není jasné, jak s ní naložit. Například v případě třídy `SqliteReader` není jasné, jak hlásit chybu uživateli programu.

Pokus o vyvolání pomocí přímo pomocí `throw` se však nesetká s úspěchem. Následující výpis vychází z předchozího. Přidává ještě další blok `try-catch`. Ten představuje klientský kód. Původní, vnitřní blok `try-catch` představuje kód, ze kterého nemůžeme vyvolat výjimku, avšak chceme ji zaznamenat.

```
1 try {
2     Base* a = 0;
3     try {
4         throw Derived();
5       }
6     catch (Base& e) {
7         a = e.clone();
8       }
9     if (a) {
10        throw *a;
```

```
11          delete a;
12        }
13    }
14 catch (Derived& e) {
15      std::cout << "Derived exception\n"; // unused
16    }
17 catch (Base& e) {
18      std::cout << "Base exception\n"; // prints: Base
                              exception
19    }
```

Důvodem proč byla zachycena základní výjimka a ne odvozená je ten, že příkaz `throw` způsobí volání kopírovacího konstruktoru. Překladač však nezná skutečný typ objektu a použije kopírovací konstruktor základní třídy. Dojde tedy k ořezání objektu (viz 5.1).

Je tedy nutné výjimku nejen polymorfně zkopírovat, ale i vyvolat. Kvůli tomu je potřeba zavést další virtuální metodu. Ta je v mnohém podobná klonovací metodě. Vhodné jméno je `raise`. Musí ji implementovat každá třída z hierarchie výjimek. Implementace je pro každou třídu stejná.

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

Pro opětovné vyvolání výjimky pak použijeme namísto příkazu `throw` metodu `raise`. Řádek číslo 10 výpisu, který demonstroval opětovné vyvolávání výjimek, se změní, jak je ukázáno na následujícím řádku kódu.

```
a->raise();
```

Výjimka, která bude vyvolána, bude mít správný typ, tj. typ který odpovídá původní, dříve zachycené a uložené výjimce.

## 5.3 Implementace ve třídě `SqliteReader`

V *gama-local* je jedna základní třída, ze které se odvozují všechny ostatní třídy výjimek[15]. Tou třídou je `GNU_gama::Exception::base`. Identifikátory `GNU_gama` a

[15]Stejná třída je základní i pro výjimky knihovny *Matvec*

`Exception` označují prostory jmen[16].

Tato třída dědí z třídy `std::exception`, základní třídy výjimek ve standardní knihovně. To znamená, že stejně jako `std::exception` má virtuální destruktor a virtuální metodu `what` a je tedy polymorfní třídou. Dále deklaruje další dvě virtuální metody, metodu `clone` a metodu `raise`. Obě metody jsou čistě virtuální a třída `Exception::base` je tedy abstraktní třídou.

Díky tomu, že je třída `Exception::base` odvozena z `std::exception`, je možné (pokud vyvstane potřeba) odchytávat všechny výjimky, tj. výjimky ze standardní knihovny i knihovny *GNU Gama*, jediným blokem `catch`. A zároveň bude dostupná metoda `what` a tedy i odpovídající chybová zpráva.

Třída výjimek, kterou používá třída `SqliteReader` a její implementace, se jmenuje `Exception::sqlitexc`.

Třída `Exception::sqlitexc` nedědí přímo z abstaktní třídy `Exception::base`, ale je potomkem třídy `Exception::string`, která přidává datový člen obsahující předávanou zprávu ve formě řetězce a dále implementuje funkci `what` tak, aby vrátila tento řetězec.

```
1  // inderr.h:
2  namespace GNU_gama { namespace Exception {
3    class base : public std::exception
4    {
5    public:
6        virtual base* clone() const = 0;
7        virtual void  raise() const = 0;
8    };
9  }}
10
11 // exception.h:
12 namespace GNU_gama { namespace Exception {
13   class string : public base {
14   public:
15       const std::string  str;
16       string(const std::string& s) : str(s) { }
```

[16]Prostor jmen `GNU_gama` nebudu dále uvádět, prostor jmen `Exception` však ano.

```
17        ~string() throw() {}
18        string*     clone() const { return new string(*this); }
19        void        raise() const { throw *this; }
20        const char* what()  const throw() { return
                          str.c_str(); }
21    };
22 }}
23
24 // sqlitereader.h:
25 namespace GNU_gama { namespace Exception {
26    class sqlitexc : public GNU_gama::Exception::string
27    {
28    public:
29        sqlitexc(const std::string& message)
30            : string(message)
31            { }
32        sqlitexc* clone() const { return new sqlitexc(*this); }
33        void      raise() const { throw *this; }
34    };
35 }}
```

Základní kód callback funkce je na následujícím výpisu. Pokud dojde k vyvolání výjimky, výjimka bude zachycena jedním ze tří bloků `catch`. Výjimky odvozené od `Exception::base` budou zachyceny prvním blokem, který provede klonování a uloží ukazatel na novou výjimku do dat třídy `SqliteReader` (přesněji do struktury `ReaderData`). Pokud výjimka není odvozena od `Exception::base`, ale je odvozena od `std::exception` (její rozhraní neumožňuje klonování), bude místo ní uložena výjimka typu `Exception::string`. Jako řetězec obsahující zprávu jí bude nastavena zpráva poskytnutá metodou `what`. Blok `catch` s výpustkou pak zachytí všechny ostatní chyby. Uložena bude opět výjimka typu `Exception::string`. Jako zpráva ji bude nastaven řetězec udávající, že došlo k neznámé (neočekávané) výjimce v callback funkci.

```
1 int readSomething(void* data, int argc, char** argv, char**)
2 {
3     ReaderData* d = static_cast<ReaderData*>(data);
```

```
4     try {
5         // ... callback's code
6         return 0;
7       }
8     catch (GNU_gama::Exception::base& e) {
9         d->exception = e.clone();
10      }
11    catch (std::exception& e) {
12        d->exception =
13                new GNU_gama::Exception::string(e.what());
14      }
15    catch (...) {
16        d->exception =
17                new GNU_gama::Exception::string("unknown");
18      }
19    return 1;
20  }
```

Funkce `exec` z implementace třídy `SqliteReader`, která obaluje volání knihovní funkce `sqlite3_exec`, zajistí opětovné vyvolání uložené výjimky. Ve výpisu je vynechán kód, který se netýká opětovného vyvolání výjimky.

```
1 void exec(sqlite3* sqlite3Handle, const std::string& query,
2           SqliteReaderCallbackType callback,
3           ReaderData* readerData)
4   {
5     char* errorMsg = 0;
6     int rc = sqlite3_exec(sqlite3Handle, query.c_str(),
                            callback,
7                                 readerData, &errorMsg);
8     if (rc != SQLITE_OK) {
9         if (readerData->exception != 0) {
10            readerData->exception->raise();
11          }
12        // ...
13      }
14  }
```

# 6 Třída SqliteReader a její implementace

Třída `SqliteReader` zajišťuje čtení dat z databáze SQLite v programu *gama-local*. Třída je umístěna v prostoru jmen `sqlite_db`, který je součástí prostoru jmen `GNU_gama::local`. V prostoru jmen `local`, který je součástí prostoru jmen `GNU_gama`, je umístěna většina tříd a funkcí, které se týkají programu *gama-local*. Definice třídy je v hlavičkovém souboru `sqlitereader.h`. Její implementace je ve zdrojovém souboru `sqlitereader.cpp`. Oba soubory lze získat společně s ostatními zdrojovými kódy projektu *GNU Gama* (viz 1), nebo samostatně z následující adresy.

`http://git.savannah.gnu.org/cgit/gama.git/tree/lib/gnu_gama/local/`

Dokumentace třídy `SqliteReader` a její implementace je v dokumentačních komentářích, které jsou součástí zdrojových kódů. Dokumentace je v anglickém jazyce. Pro vytvoření dokumentace ve formě HTML stránek a PDF dokumentu byl použit program *Doxygen*. Zkrácená verze PDF dokumentu je v příloze B.

## 6.1 Rozhraní

Rozhraní třídy tvoří konstruktor, který přebírá jeden parametr typu `std::string` reprezentující název databázového souboru, a metoda `retrieve`. Metoda `retrieve` načte konfiguraci uloženou v databázi do objektu typu `LocalNetwork`. Metoda přebírá jako parametr referenci na ukazatel na objekt typu `LocalNetwork`. Další parametr je typu `std::string` a reprezentuje jméno konfigurace, která je uložena v databázi. To, že je první parametr (nekonstantní) reference, umožňuje metodě změnit hodnotu ukazatele. K tomu dojde ve chvíli, kdy je metodě předán nulový ukazatel (`NULL`). V tomto případě bude vytvořen nový objekt třídy `LocalNetwork` na základě algoritmu, který je uložen v databázi.[17] Tím je programu *gama-local* umožněno, aby pokud uživatel nezadá algoritmus, použil ten, který je uložen v databázi, namísto toho, aby použil implicitní algoritmus. Hlavičky konstruktoru a metody `retrieve` ukazuje následující výpis.

```
SqliteReader(const std::string &fileName);
```

[17] Třída `LocalNetwork` je abstraktní. Konkrétní třídy se liší podle použitého algoritmu, takže je možné je vytvářet, jen pokud ho známe.

```
void retrieve (LocalNetwork*& lnet,
               const std::string& configuration);
```

Jako třída výjimek pro hlášení chyb vzniklých při čtení z databáze slouží třída `Exception::sqlitexc`.

## 6.2 Implementace

Ve třídě je využita technika soukromé implementace, popsaná v části 4, proto má pouze jeden datový člen, který je soukromý. Tento datový člen je typu `ReaderData`. Třída `ReaderData` je definována v souboru `sqlitereader.cpp` a je součástí prostoru jmen `sqlite_db`. Soubor `sqlitereader.h` obsahuje pouze dopřednou deklaraci třídy `ReaderData`. Třída `SqliteReader` nemá žádné soukromé metody.

Do implementace třídy patří kromě třídy `ReaderData` také funkce v bezejmenném prostoru jmen (v souboru `sqlitereader.cpp`) a především callback funkce, pomocí kterých probíhá čtení z databáze.

Třída `SqliteReader` zajišťuje spojení s databází. Přitom využívá techniku RAII (*resource acquisition is initialization*) popsanou například v [1, str. 366]. Technika spočívá v tom, že se zdroj alokuje při inicializaci objektu a uvolní se při jeho destrukci. V našem případě to znamená, že konstruktor třídy `ReaderData` otevírá databázi (funkcí `sqlite3_open`) a destruktor databázi uzavírá (funkcí `sqlite3_close`). A vzhledem k tomu, že konstruktor v případě neúspěšného pokusu o otevření databáze vyvolá výjimku, nemůže se stát, že bychom nevědomky pracovali s neotevřenou databází. Podobně díky tomu, že uzavření databáze proběhne automaticky v destruktoru, nemůže se stát, že bychom databázi nezavřeli. K uzavření dojde i v případě, že byla vyvolána výjimka. Destruktor bude zavolán automaticky a díky němu i funkce `sqlite3_close`. Ukazatel na databázový objekt obsahuje samozřejmě třída `ReaderData`.

Protože je ke komunikaci s databází použito SQLite C/C++ API (rozhraní s callback funkcemi), bylo třeba dbát na správné propojení jazyků C a C++, kterým se zabývá část 3. Konkrétní implementace pro třídy `SqliteReader` a `ReaderData` je podrobněji rozebrána v části 3.3.3.

Názvy callback funkcí začínají `sqlite_db_`, aby se zabránilo konfliktu jmen a

případně špatnému požití. Prefix `sqlite_db_` je odvozen z názvu prostoru jmen, ve kterém jsou umístěny třídy `SqliteReader` a `ReaderData`. [18]

Obecné informace o callback funkcích používaných v SQLite C/C++ API jsou uvedeny v části 2.2. Ošetření výjimek vzniklých v callback funkcích je uděláno tak, jak popisuje část 5.3. Výjimky zachycené v callback funkcích jsou ukládány ve třídě `ReaderData` a znovu vyvolávány ve funkci `exec`. Ta je obálkou kolem funkce `sqlite3_exec`.

Všechny callback funkce v implementaci třídy `SqliteReader` očekávají, že jim pomocí ukazatele na `void` bude předán objekt typu `ReaderData`. Každá callback funkce však očekává objekt `ReaderData` v jiném stavu. Podle toho lze callback funkce rozdělit do dvou skupin. V první skupině jsou následující funkce (uveden je jen název – návratovou hodnotu a parametry mají všechny callback funkce stejné).

```
sqlite_db_readConfigurationInfo
sqlite_db_readConfigurationText
sqlite_db_readPoints
sqlite_db_readClusters
```

Funkce zapisují různé hodnoty do objektu třídy `LocalNetwork`, na který je ve třídě `ReaderData` ukazatel. Funkce proto očekávají, že tento ukazatel je platný.

Ve druhé skupině jsou funkce, které zapisují údaje do určitých objektů v objektu třídy `LocalNetwork`. Příkladem takových objektů můžou být objekty typu `StandPoint` či `CovMat`. Callback funkce samy o sobě však neví, do jakého objektu mají zapisovat. Proto je nutné předat jim tuto informaci. K tomu dobře poslouží ukazatel uložený v objektu třídy `ReaderData`, který se callback funkcím předává. Tento ukazatel musí nastavit volající funkce `exec` (který je vlastně nepřímým volajícím callback funkce). Nabízí se otázka, proč nepředávat do callback funkcí pomocí ukazatele na `void` přímo ukazatel na zpracovávaný objekt. Odpověď je prostá. Nebylo by totiž možné v callback funkci ukládat výjimky, protože by nebylo by kam. Následuje seznam callback funkcí, které jsou v popsané skupině.

[18] Callback funkce, které jsou používané v SQLite C/C++ API a mají tedy C linkování, sice mohou být umístěny v prostoru jmen, ale nezabrání to případnému konfliktu jmen.

```
sqlite_db_readObservations
sqlite_db_readVectors
sqlite_db_readCoordinates
sqlite_db_readHeightDifferences
sqlite_db_readCovarianceMatrix
```

## 6.3 Integrace do *gama-local*

Aby mohla být třída `SqliteReader` začleněna do projektu *GNU Gama* konkrétně v programu *gama-local*, bylo nutné provést provést určité úpravy ve stávajícím kódu.

V první řadě musela být vylepšena hierarchie výjimek. Především byly přidány metody `clone` a `raise`, pro umožnění polymorfní práce s výjimkami. Jak vypadá ta část hierarchie výjimek, která se týká třídy `SqliteReader`, je uvedeno v části 5.3.

Musela být také změna funkce `main` programu *gama-local*, která se nachází v souboru `bin/gama-local.cpp`. Byl přidán kód, který na základě parametrů předaných programu v příkazové řádce načte údaje z databáze pomocí třídy `SqliteReader`, nebo načte údaje z XML souboru, tak jako tomu bylo v dřívějších verzích.

Do projektu byla také přidána jednoduchá tovární metoda, chcete-li funkce, `newLocalNetwork`. Ta na základě názvu algoritmu vytvoří nový objekt, který je instancí jedné ze tříd odvozených ze třídy `LocalNetwork`. Právě na základě názvu algoritmu je rozhodnuto, jaká konkrétní třída bude použita. K zavedení tovární metody vedl fakt, že v některých případech musí být nový objekt vytvořen ve funkci `main` programu *gama-local* a jindy musí být vytvořen v jedné z metod třídy `SqliteReader`. Deklarace tovární metody `newLocalNetwork` následuje.

```
LocalNetwork* newLocalNetwork(std::string algorithm = "");
```

Parametrem je název algoritmu. Názvy se shodují s názvy, které jsou použity v parametrech příkazové řádky. Pokud je řetězec prázdný nebo je název neplatný, je řetězec nahrazen řetězcem `"gso"`. Deklarace tovární metody `newLocalNetwork` je v souboru `newnetwork.h` a je umístěna do prostoru jmen `GNU_gama::local`. Definice je v souboru `newnetwork.cpp`. Výhodou tohoto řešení je, že pro vytváření objektů tříd odvozených ze třídy `LocalNetwork` již není potřeba vkládat definice všech těchto tříd. Další výhodou je to, že kód, který rozhoduje o vytvoření konkrétní třídy na základě názvu algoritmu, je na jednom místě a zároveň může být zavolán

z více funkcí. Toto byl vlastně důvod zavedení jednoduché tovární metody. Nutno ještě poznamenat, že volající kód nemusí znát deklarace konkrétních tříd, protože používá tovární metodu.

Jak již bylo uvedeno v části 2, projekt *GNU Gama* se snaží o minimální závislost na jiných knihovnách. Projekt *GNU Gama* používá pro sestavení na operačních systémech Unixového typu sestavovací systém *GNU Autotools*[19]. Tento systém umožňuje sestavit projekt *GNU Gama* i v případě, že knihovna SQLite není na daném počítači nainstalována. Při samotném sestavování zdrojových kódů se pak podpora databáze SQLite vůbec nevytvoří a to díky podmíněnému překladu. Všechny části kódu, které se týkají podpory databáze, jsou mezi direktivami preprocesoru `#ifdef GNU_GAMA_LOCAL_SQLITE_READER` a `#endif`. Jedná se o obsah hlavičkového souboru `sqlitereader.h`, zdrojového souboru `sqlitereader.cpp` a některé části souboru `bin/gama-local.cpp`. V případě, že není definováno makro (identifikátor) `GNU_GAMA_LOCAL_SQLITE_READER`, nebudou části týkající se podpory SQLite databáze do překladu zahrnuty. O tom, zda má být makro definováno rozhodne právě systém *GNU Autotools*.

Většinu kódu potřebného k integraci třídy `SqliteReader` napsal autor projektu *GNU Gama* a vedoucí této práce, prof. Ing. Aleš Čepek, CSc..

# 7 Testování

Testování je důležité pro každý program. Pro program *gama-local* je důležitější o to, že si uživatel programu většinou nemůže nijak ověřit, že výsledky, které mu program poskytl, jsou správné. Vývoj třídy `SqliteReader` – stejně jako vývoj celého projektu *GNU Gama* – probíhá pod systémy z rodiny *GNU/Linux*. Díky lze při testování využít řadu možností, které tyto systémy nabízí.

Kontrola správnosti byla provedena dvěma způsoby. Jeden využívá program *diff* (viz 7.2) a druhý využívá program *gama-local-cmp* (viz 7.3). Přímo během vývoje se využily možnosti kompilátoru *GCC* (viz 7.1). Dále se samozřejmě také používaly, ne příliš uznávané, avšak hojně využívané, orientační testy pomocí textových výpisů všeho druhu.

[19] `http://autotoolset.sourceforge.net/`

## 7.1 Kontroly pomocí kompilátoru *GCC*

Kompilátory C++ nejen upozorňují na chyby, kvůli kterým nelze kód sestavit, ale jsou také schopny poskytnout varování, která mohou upozornit na problematický kód.[20] Jak je uvedeno v [4], v C++ je velmi vhodné používat kontroly během kompilace. Ty jsou možné například proto, že C++ je staticky typovaný jazyk. Autoři toto shrnují do dvou pravidel (1 a 14).

> *Kompilujte bez varování i při vysoké citlivosti kompilátoru.*
>
> *Chyby při sestavování jsou lepší než chyby při běhu.*

Kompilátor *GCC* nabízí řadu nastavení, kterými lze řídit kontroly během kompilace. Lze určit, v jakých případech má kompilátor podat varovnou zprávu, a lze také určit, jaká varování se mají stát chybami, čímž si lze na programátorovi vynutit dodržení dodržení prvně jmenovaného pravidla.

Při vývoji třídy `SqliteReader` byla použita celá řada nastavení kompilátoru *GCC*. Jejich kompletní seznam je uveden v příloze C. Tato nastavení vyvolají řadu varovných hlášení, která často pouze upozorňují na nedodržení jistých dobrých zvyků (například nastavení `-Weffc++`).

Projekt *GNU Gama* není kompilován s tak citlivým nastavením kompilátoru, jako bylo použito pro vývoj třídy `SqliteReader`. Nižší citlivost kompilátoru však není nutně chybou. Například výše zmíněné nastavení `-Weffc++` nutí v některých případech explicitně inicializovat objekt typu `std::string` prázdným řeťezcem (`""`), ačkoli je tato inicializace implicitní. Použití zmiňovaných nastavení při kompilaci projektu *GNU Gama* by způsobilo velké množství varovných hlášení, ve kterých by se snadno ztratila varovná hlášení způsobená novým, vyvíjeným kódem. Avšak třída `SqliteReader` byla vyvíjena v samostatném projektu a projekt *GNU Gama* byl použit již zkompilovaný v podobě knihovny. To by však samo o sobě nestačilo ke skrytí všech varování způsobených stávajícím kódem projektu *GNU Gama*, protože vždy je třeba vkládat hlavičkové soubory. Nejnovější verze kompilátoru *GCC* (4.6) umožňuje skrýt varování z hlavičkových souborů pomocí pomocí direktivy `#pragma`.

[20] Dnes už podobné schopnosti mají i editory, například program *Qt Creator*, který jsem používal při vývoji. Program *Qt Creator* však nenabízí takové možnosti kontroly jako kompilátor *GCC*.

Ve verzi 4.4, kterou používám já, tato možnost není, avšak je možné připojit při kompilaci hlavičkové soubory jako systémové, což skryje varování z takto připojených hlavičkových souborů. Hlavičkové soubory se připojí jako systémové tak, že místo obvyklého `-I` se před cestu k souborům napíše `-isystem`, v mém případě tedy `-isystem../gama/lib`. Další možností, jak se zbavit varovných hlášení je filtrovat hlášení pomocí programu *grep*. Toto řešení však nefunguje tak dobře, jako když se použije přímo kompilátor.

V manuálu ke kompilátoru *GCC* [3] je pro programy v C a C++ doporučeno toto nastavení:

```
-ansi -pedantic -Wall -Wextra
-Wconversion -Wshadow -Wcast-qual -Wwrite-strings
```

Z těchto nastavení zde pro zajímavost vysvětlím pouze dvě. Nastavení `-Wconversion` způsobí varování, při konverzi z `double` na `int`. A nastavení `-Wshadow` způsobí varování, když lokální proměnná zakryje svým jménem jinou proměnou. Ještě je vhodné poznamenat, že nastavení `-Wextra` je stejné jako nastavení `-W`. Označení `-W` je však zastaralé. Jak již bylo uvedeno, při vývoji třídy `SqliteReader` bylo použito více nastavení (viz příloha C). Nastavení doporučená manuálem jsou však ta nejužitečnější.

Nicméně žádný kompilátor nemůže zachytit všechny chyby, proto je vždy nutné provádět testování.

## 7.2 Testování pomocí programu *diff*

Program *diff* je jedním z malých jednoúčelových programů používaných na unixových systémech, jinými slovy patří mezi takzvané unixové utility. Program umožňuje porovnat mezi sebou dva textové soubory. Testování programu *gama-local* spočívá v tom, že se programem *gama-local* spočítá síť. Při tom se je jako výstupní formát nastaví prostý textový soubor. Výpočet se provede dvakrát. Jednou se data čtou z XML souboru a jednou z databáze SQLite. Z každého výpočtu je tedy jeden textový soubor. Tyto soubory se následně porovnají programem *diff*.

Pro hromadné testování byl vytvořen skript pro *bash*. Ten obsahuje i příkazy, které vytvoří a naplní SQLite databázi. Jako testovací data byla použita sbírka

příkladů, která je součástí projektu *GNU Gama*, s výjimkou těch příkladů, které způsobují problémy při vyrovnání.[21]

Porovnání programem *diff* má však jisté nevýhody. Nevýhodu, která se projevuje nejčastěji, je to, že porovnává jednotlivé znaky, nikoli hodnoty v souborech. V textovém výstupu se však někdy stává, že nějaká veličina má sice v obou souborech hodnotu nula, ale tato nula má v jednom souboru znaménko kladné a ve druhém záporné (je to dáno tím, jak funguje textový výstup v jazyce C++ a v programu *gama-local*). Program *diff* takovéto soubory označí jako rozdílné a výsledkem testu je, že se výpočty lišily, ačkoli byly stejné.

Dále se může stát, v důsledku numerického šumu, že výsledné hodnoty nebudou v textových souborech ve stejném pořadí. V souborech se bude lišit pořadí řádků a program *diff* je opět označí jako rozdílné, i když se výsledky výpočtů neliší.

Další nevýhoda plyne z použití textového formátu jako výstupu z programu *gama-local*. Hodnoty v tomto výstupu jsou totiž zaokrouhlené, a tak je možné, že nebudou odhaleny malé rozdíly ve výsledných hodnotách. Mohlo by se zdát, že by problém byl vyřešen použitím souborů ve formátu XML. To by ovšem způsobilo jen další problémy. Hodnoty v XML souborech nejsou totiž vůbec zaokrouhlené. To způsobí, že číslo, které můžeme vzhledem k přesnosti, s jakou probíhá výpočet v programu a s jakou se počítá v geodézii, považovat za nulu, bude vyjádřeno dvěma zcela odlišnými čísly. Řádky s těmito čísly program *diff* samozřejmě označí jako rozdílné. Důsledkem by opět bylo označení shodných výpočtů za neshodné.

I přes jmenované nevýhody je porovnání výstupních souborů v textovém formátu pomocí programu *diff* účinnou testovací technikou a bylo donedávna jediným způsobem, jakým se provádělo hromadné testování programu *gama-local*.

Program *diff* našel rozdíly v mnoha textových souborech, avšak většina těchto rozdílů byla dána pouze různým zápisem hodnot (viz výše).

V několika případech se však jednalo o skutečné rozdíly ve výsledcích. Následující část 7.3 tyto rozdíly rozebírá.

[21] Příklady lze získat, stejně jako zdrojové kódy, ze stránek projektu [14]. Problematickým konfiguracím se věnuje práce [6].

## 7.3 Testování pomocí programu *gama-local-cmp*

Program *gama-local-cmp* je součástí diplomové práce [6]. Z ní a také od jejího autora Ing. Gabriela Györiho jsem čerpal znalosti o funkcionalitě programu a způsobu použití.

Program *gama-local-cmp* umožňuje porovnat dva XML soubory s výsledky vyrovnání z programu *gama-local*. Program porovnává hodnoty načtené z XML souborů a určuje jejich rozdíly. S pomocí programu *gama-local-cmp* je možné sledovat posuny v geodetických sítí, avšak stejně tak je vhodný i k testování výsledků vyrovnání programem *gama-local*. Lze jej využít ke zjištění rozdílů ve výpočtu různými algoritmy nebo různými verzemi programu *gama-local* (tak byl použit i ve zmiňované diplomové práci [6]). Zde je program použit pro zjištění rozdílů ve vyrovnání, kdy jednou je vstupem XML soubor a jednou data z databáze. Program *gama-local-cmp* vypisuje všechny rozdíly, které jsou větší než 1e-09.

Samotné testování probíhalo stejně jako s programem *diff* (viz 7.2). Jen skript pro *bash* musel být částečně pozměněn.

Testy ukázaly jisté rozdíly mezi výsledky vyrovnání, kde vstupem byl XML soubor, a vyrovnání, kde vstupem byla databáze SQLite. Tyto rozdíly byly pouze u prvků kovarianční matice. Výsledné souřadnice a vyrovnaná měření včetně jejich doplňujících údajů se ve všech testovaných souborech shodovala. Absolutní hodnoty rozdílů byly řádu 1e-04 a menší. Následné zvýšení přesnosti výpisu prvků kovarianční matice při generovaní výstupního XML souboru[22] rozdíly mezi hodnotami zmenšilo tak, že byly nejvýše řádu 1e-06, většinou však 1e-09. Po celou dobu testování byla přesnost výpisů nastavena na 20 desetinných míst, což je zbytečně vysoká hodnota, avšak vylučuje chybu při výpisu.

Aby byl při dalším testování práce s databází vyloučen vliv nového kódu (tj. třídy `SqliteReader`), byl použit program na převod údajů uložených v databázi SQLite do vstupní dávky XML. Tento program se používá pouze pro účely testování a jmenuje se *sql2xml*. Testování spočívalo v tom, že nejprve se převedl testovaná konfigurace uložená v XML souboru do SQL dávky (převod zajišťuje program *gama-local-xml2sql*). Touto dávkou byla následně naplněna databáze. Z databáze byla konfigurace přenesena zpět do XML programem *sql2xml*. Obě vstupní dávky byly

[22] Změna se týká třídy `LocalNetworkXML`, metody `write` v souboru `localnetowork.cpp`.

postupně vyrovnány programem *gama-local*. XML soubory s výsledky vyrovnání byly porovnány pomocí programu *gama-local-cmp*.

Porovnání opět ukázalo rozdíly. V dokumentace databáze SQLite [12] je uvedeno, byť poněkud v jiné souvislosti, že přesnost čísel s plovoucí desetinou čárkou (`REAL`) je 15 platných cifer.[23] Také pokusy, které jsem provedl s SQLite C/C++ API a v příkazové řádce SQLite, ukázaly, že při uložení je číslo zaokrouhleno na 15 platných cifer. To ukázalo na to, že rozdíly by mohly vznikat při ukládání hodnot do databáze. Zkoumání jednotlivých kroků při převádění dat toto také potvrdilo.

Pro zjišťování chybných převodů jsem vytvořil konfiguraci, kde je síť tvořena jen třemi body (dva jsou známé, jeden neznámý). Aby byl vyčerpán počet platných cifer, tak jsem jako souřadnice známých bodů jsem zvolil velké hodnoty. Zde je ukázka jednoho bodu.

```
<point id='1' x='600154980.484654321'
              y='100644898.590654321' fix='xy'/>
```

Čísla mají 18 platných cifer. Do databáze se však uloží pouze 15, zbytek se ztratí. Tato ořezaná čísla se při převodu uloží do XML souboru (ukázka následuje).

```
<point id='1' x='600154980.484654'
              y='100644898.590654' fix='xy'/>
```

Výsledky vyrovnání programem *gama-local* se kvůli rozdílů ve vstupních souborech také liší. Program *gama-local-cmp* v tomto konkrétním případě zjistil rozdíly v řádu 1e-07. Rozdíly jsou v souřadnicích vypočteného bodu a samozřejmě také v souřadnicích pevných bodů.

Na základě výše uvedených poznatků lze rozdíly výsledků vyrovnání (při čtení údajů z XML a při čtení údajů z databáze) vysvětlit tím, že při ukládání čísel do databáze SQLite dochází k zaokrouhlení[24]. To se následně projeví jak při načítání údajů pomocí třídy `SqliteReader`, tak i při použití souborů získaných programem *sql2xml*.

[23] K přesnosti uložení čísel je nutné ještě poznamenat, že způsob uložení čísel v C++ je závislý na implementaci. Norma C++ [2] pouze zaručuje jisté minimální požadavky, které musí splňovat daný typ.

[24] Jak ukázaly testy, databáze SQLite čísla při ukládání neořezává, ale zaokrouhluje.

Rozdíly v souřadnicích se však projevily pouze v testovacím souboru vytvořeném pro tento účel. Ve skutečných konfiguracích (ze sbírky příkladů projektu *GNU Gama*) se však rozdíly ve vypočtených hodnotách projevily pouze u prvků kovarianční matice. Rozdíly jsou malé a tak nejsou výsledky vyrovnání znehodnoceny.

K velikosti rozdílů prvků kovarianční matice je ještě nutné poznamenat, že program *gama-local-cmp* počítá a posuzuje absolutní rozdíl dvou prvků. Avšak hodnoty prvků v kovarianční matici jsou řádově různé, v závislosti na dané konfiguraci (většinou od 1e-02 do 1e+03). Posuzování přesnosti tím způsobem, že se prvky odečtou a rozdíl porovná s hodnotou 1e-09, může v případě malých hodnot prvků vést k přehlédnutí chyby a v případě velkých hodnot naopak k označení chyby tam, kde nenastala.

# Závěr

Cílem této bakalářské práce bylo rozšířit program *gama-local* o funkcionalitu, která umožní čtení dat přímo z databáze SQLite. Data přečtená z databáze jsou ukládána přímo do datových struktur, které program *gama-local* použije pro vyrovnání lokální geodetické sítě. Důležité bylo implementovat čtení dat z databáze na takové úrovni, aby novou funkcionalitu bylo možné zapojit do projektu *GNU Gama*.

Pro čtení dat z databáze SQLite bylo použito nativní rozhraní (SQLite C/C++ API). Konkrétně ta část rozhraní, kde se využívají callback funkce. Rozhraní je určeno jak pro jazyk C, tak i pro jazyk C++. Jeho použití má jistá specifika, která plynou z rozdílů obou jazyků. Tato práce se jim zevrubně věnuje.

Novou funkcionalitu bylo nutné otestovat. Pro účely testování bylo vytvořeno několik skriptů pro *bash*, které využívají programy *diff* a *gama-local-cmp*. Jako testovací data byla použita téměř celá sbírka příkladů, které jsou součástí projektu *GNU Gama*. Testy probíhaly tak, že se porovnávaly výsledky spočtené na základě dat ze souboru XML s výsledky spočtenými na základě dat z databáze SQLite.

Testy ukázaly, že při načítání data z databáze nedochází k žádným chybám, které by způsobily rozdíly ve vyrovnaných souřadnicích a vyrovnaných měřeních. Program *gama-local-cmp* však odhalil rozdíly mezi prvky kovarianční matice vyrovnaných neznámých. Další testy ukázaly, že rozdíly jsou způsobené ukládáním dat do databáze SQLite. Způsob ukládaní čísel je daný databází SQLite a nelze ho tudíž změnit. Zjištěné rozdíly se týkají směrodatných odchylek vyrovnaných souřadnic, nikoli souřadnic samotných. Tyto rozdíly jsou malé a neznehodnocují výsledky vyrovnání. Vzhledem k tomu, lze považovat rozšíření programu *gama-local* za plně funkční.

Díky možnosti čtení údajů z databáze SQLite mají uživatelé programu *gama-local* alternativu k XML souboru. Mohou tak použít to, co je pro jejich práci výhodnější.

Již během vývoje se ukázalo, že čtení dat přímo z databáze by pro uživatele mohlo být velkou výhodou. Dalším krokem tedy jistě bude rozšíření podpory databáze SQLite v programu *gama-local* o zápis výsledků vyrovnání do databáze. Při tom bude výhodné použít opět SQLite C/C++ API. Předtím však bude nutné navrhnout databázové schéma pro výsledky vyrovnání, které bude muset být – stejně jako schéma pro vstupní údaje – v souladu s potřebami projektu *QGama*.

# Použité zdroje

[1] STROUSTRUP, Bjarne. *The C++ Programming Language – Special Edition*. AT&T Labs, Florham Park, New Jersey. United States of America: Addison-Wesley, 2000. 1020 s. ISBN 0-201-70073-5.

[2] 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 s.

[3] GOUGH, Brian J. *An Introduction to GCC – for the GNU Compilers gcc and g++*. Foreword by Richard M. Stallman. Network Theory Limited, United Kingdom. Revised August 2005. ISBN 0954161793. URL: <http://www.network-theory.co.uk/docs/gccintro/>

[4] SUTTER, Herb; ALEXANDRESCU, Andrei. *101 programovacích technik*. První vydání. Brno: Zoner Press, 2005. 232 s. ISBN 80-86815-28-5.

[5] ČEPEK, Aleš. *Informatika: Úvod do C++*. První vydání. Praha: Nakladatelství ČVUT, 2004. 265 s. ISBN 80-01-03074-1

[6] GYÖRI, Gabriel. *Analýza a kontrola XML výsledků vyrovnania GNU Gama*. Praha, 2011. 112 s. Diplomová práce. ČVUT v Praze, Fakulta stavební.

[7] *GNU General Public License* [online]. Version 3, 29 June 2007. Free Software Foundation, Inc. 51 Franklin Street, Suite 500 Boston, MA 02110-1335 USA [cit. 2011-03-12]. URL: <http://www.gnu.org/licenses/gpl.html>

[8] W3C. *Extensible Markup Language (XML)* [online]. Last modified 2011-03-08. c1996-2003 World Wide Web Consortium MIT/CSAIL in USA [cit. 2011-04-03]. URL: <http://www.w3.org/XML/>

[9] ČEPEK, Aleš. *Manuál k programu GNU Gama* [online]. 2010-08-22 [cit. 2011-03-12]. URL: <http://www.gnu.org/software/gama/manual/gama.pdf>

[10] *An Introduction To The SQLite C/C++ Interface* [online]. Modified 2011-03-28 [cit. 2011-03-30]. URL: <http://www.sqlite.org/cintro.html>

[11] *C/C++ Interface For SQLite Version 3* [online]. Modified 2011-04-17 [cit. 2011-04-18]. URL: <`http://www.sqlite.org/capi3ref.html`>

[12] *Datatypes In SQLite Version 3* [online]. Modified 2011-04-17 [cit. 2011-04-19]. URL: URL: <`http://www.sqlite.org/datatype3.html`>

[13] Qt Development Frameworks. *Qt – Cross-platform application and UI framework* [online]. c2008-2011 Nokia Corporation [cit. 2011-04-18]. URL: <`http://qt.nokia.com/`>.

[14] ČEPEK, Aleš. *GNU Gama* [program]. Edition 1.10, c2009-. URL: <`http://www.gnu.org/software/gama/`>

[15] NOVÁK, Jiří. *QGama* [program]. Last update 2010-10-18. URL: <`http://sourceforge.net/projects/qgama/`>

# Seznam použitých zkratek

| | |
|---|---|
| **SQL** | Structured Query Language |
| **XML** | Extensible Markup Language |
| **HTML** | HyperText Markup Language |
| **PDF** | Portable Document Format |
| **API** | Application Programming Interface |
| **GCC** | GNU Compiler Collection |
| **GPL** | General Public License |
| **LGPL** | Lesser General Public License |
| **RAII** | Resource Acquisition Is Initialization |
| **PIMPL** | Private Implementation, Pointer to Implementation |

# Seznam příloh

# A Databázové schéma gama-local-schema.sql

```
/*
    GNU Gama -- adjustment of geodetic networks
    Copyright (C) 2010 Ales Cepek <cepek@gnu.org>, 2010 Jiri Novak
    <jiri.novak@petriny.net>, 2010 Vaclav Petras <vaclav.petras@fsv.cvut.cz>

    This file is part of the GNU Gama C++ library.

    This library is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 3 of the License, or
    (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  $
*/


create table gnu_gama_local_configurations (
   conf_id   integer primary key,
   conf_name varchar(60) not null unique,
   sigma_apr double precision default 10.0 not null check (sigma_apr > 0),
   conf_pr   double precision
               default 0.95 not null check (conf_pr > 0 and conf_pr <1),
   tol_abs   double precision default 1000 not null check (tol_abs > 0),
   sigma_act varchar(11)
               default 'aposteriori' not null
               check (sigma_act in ('apriori', 'aposteriori')),
   update_cc varchar(3) default 'no' not null check (update_cc in ('yes', 'no')),
   axes_xy   varchar(2)
               default 'ne' not null
               check
               (axes_xy in ('ne', 'sw', 'es', 'wn', 'en', 'nw', 'se', 'ws')),
   angles    varchar(12)
               default 'right-handed' not null
               check (angles in ('left-handed', 'right-handed')),
   epoch     double precision default 0.0 not null,
   algorithm varchar(12)
               default 'svd' not null
               check (algorithm in ('svd', 'gso', 'cholesky', 'sm-env')),
   ang_units int default 400 not null check (ang_units in (400, 360)),
   latitude  double precision default 50 not null,
   ellipsoid varchar(20)
```

```
);

create table gnu_gama_local_descriptions (
   conf_id   integer references gnu_gama_local_configurations,
   indx      integer check (indx >= 1),
   text      varchar(1000) not null,
   primary key (conf_id, indx)
);

create table gnu_gama_local_points (
   conf_id   integer references gnu_gama_local_configurations,
   id        varchar(80),
   x         double precision,
   y         double precision,
   z         double precision,
   txy       varchar(11) check (txy in ('fixed', 'adjusted', 'constrained')),
   tz        varchar(11) check (tz  in ('fixed', 'adjusted', 'constrained')),
   primary key (conf_id, id)
);

create table gnu_gama_local_clusters (
   conf_id   integer references gnu_gama_local_configurations,
   ccluster  integer check (ccluster > 0),
   dim       integer not null check (dim > 0),
   band      integer not null,
   tag       varchar(18)
                not null
                check
                (tag in ('obs', 'coordinates', 'vectors', 'height-differences')),
   check (band between 0 and dim-1),
   primary key (conf_id, ccluster)
);
-- upper triangular variance-covariance band-matrix (0 <= bandwidth < dim)

create table gnu_gama_local_covmat (
   conf_id   integer,
   ccluster  integer,
   rind      integer check (rind > 0),
   cind      integer check (cind > 0),
   val       double precision not null,
   foreign key (conf_id, ccluster) references gnu_gama_local_clusters,
   primary key (conf_id, ccluster, rind, cind)
);

create table gnu_gama_local_obs (
   conf_id   integer,
   ccluster  integer,
   indx      integer check (indx > 0),
   tag       varchar(10)
                check
                (tag in
```

```
                ('direction', 'distance', 'angle', 's-distance', 'z-angle', 'dh')),
   from_id    varchar(80) not null,
   to_id      varchar(80) not null,
   to_id2     varchar(80),
   val        double precision not null,
   stdev      double precision,
   from_dh    double precision,
   to_dh      double precision,
   to_dh2     double precision,
   dist       double precision, -- dh dist
   rejected   integer default 0 not null,
   primary key (conf_id, ccluster, indx),
   foreign key (conf_id, ccluster) references gnu_gama_local_clusters,
   check (tag <> 'angle' or to_id2 is not null),
   check (tag = 'dh' or (tag <> 'dh' and dist is null))
);

create table gnu_gama_local_coordinates (
   conf_id    integer,
   ccluster   integer check (ccluster > 0),
   indx       integer check (indx > 0),
   id         varchar(80),
   x          double precision,
   y          double precision,
   z          double precision,
   rejected   integer default 0 not null,
   foreign key (conf_id, ccluster) references gnu_gama_local_clusters,
   primary key (conf_id, ccluster, indx)
);

create table gnu_gama_local_vectors (
   conf_id    integer,
   ccluster   integer check (ccluster > 0),
   indx       integer check (indx > 0),
   from_id    varchar(80),
   to_id      varchar(80),
   dx         double precision,
   dy         double precision,
   dz         double precision,
   from_dh    double precision,
   to_dh      double precision,
   rejected   integer default 0 not null,
   foreign key (conf_id, ccluster) references gnu_gama_local_clusters,
   primary key (conf_id, ccluster, indx)
);
```

# B Dokumentace třídy `SqliteReader`

Dokumentace třídy `SqliteReader` je v dokumentačních komentářích ve zdrojových kódech, zdokumentováno bylo rozhraní i implementace. Dokumentace je generována pomocí nástroje *Doxygen*, který z dokumentačních komentářů generuje dokumentaci (referenční manuál) v několika formátech (HTML, PDF, XML, ...). Dokumentačními komentáři se rozumí speciálně označené běžné komentáře, které obsahují navíc ještě formátovací a jiné značky.

Tato příloha obsahuje zkrácenou verzi dokumentace ve formátu PDF. Vynechány byly například callback funkce, jejich jména však dostatečně vypovídají o jejich účelu. Navíc jsou callback funkce popsány v hlavním textu.

# 4 Namespace Documentation

## 4.1 anonymous_namespace{sqlitereader.cpp} Namespace Reference

### Functions

- void **exec** (sqlite3 *sqlite3Handle, const std::string &query, **SqliteReaderCallbackType** callback, **ReaderData** *readerData)
- double **ToDouble** (const char *s, const std::string &m=**T_gamalite_conversion_to_double_failed**)
- int **ToInteger** (const char *s, const std::string &m=**T_gamalite_conversion_to_integer_failed**)

### Variables

- const char * **T_gamalite_database_not_open**
- const char * **T_gamalite_invalid_column_value**
- const char * **T_gamalite_conversion_to_double_failed**
- const char * **T_gamalite_conversion_to_integer_failed**
- const char * **T_gamalite_unknown_exception_in_callback**
- const char * **T_gamalite_stand_point_cluster_with_multi_dir_sets**
- const char * **T_gamalite_configuration_not_found**

### 4.1.1 Function Documentation

#### 4.1.1.1 void anonymous_namespace{sqlitereader.cpp}::exec (sqlite3 * *sqlite3Handle*, const std::string & *query*, SqliteReaderCallbackType *callback*, ReaderData * *readerData*)

A C++ wrapper around `sqlite3_exec` function.

**For internal use only.**

Connection to database has to be open. If the *callback* is `NULL` pointer, then no callback function is called (result rows are ignored). If callback requests query execution abort (by returning non-zero value), no other callbacks is called and exception is thrown (`sqlite3_exec` returns a value which differs from `SQLITE_OK`, see also <`http://www.sqlite.org/capi3ref.html#SQLITE_ABORT`>).

If callback stores pointer to an exception to **GNU_gama::local::sqlite_db::ReaderData::exception** (p. 9) and callback requests query execution abort, exception will be rethrown. If **GNU_gama::local::sqlite_db::ReaderData::exception** (p. 9) is `NULL` pointer, **GNU_gama::Exception::sqlitexc** (p. 11) will be thrown with SQLite error message.

**Parameters**

***sqlite3Handle*** pointer to `struct sqlite3`

***query*** query string

***callback*** pointer to callback function

***readerData*** pointer to `struct` ReaderData

**Exceptions**

***GNU_gama::Exception::sqlitexc*** (p. ***11***) if error occurs when reading from database

***GNU_gama::Exception::base*** if is error occurred by something another -- it depends on callback It can also throw any other exception derived from this class.

Callback functions are expected to handle exceptions like this:

```
\\ ... try block
catch (GNU_gama::Exception::base& e)
    {
        d->exception = e.clone();
    }
catch (std::exception& e)
    {
        d->exception = new GNU_gama::Exception::string(e.what());
    }
catch (...)
    {
        d->exception = new GNU_gama::Exception::string("unknown");
    }
return 1;
```

**See also**

**GNU_gama::local::sqlite_db::ReaderData** (p. 6), **SqliteReaderCallbackType** (p. 14), **GNU_gama::Exception::sqlitexc** (p. 11)

Referenced by GNU_gama::local::sqlite_db::SqliteReader::retrieve(), and sqlite_db_readClusters().

#### 4.1.1.2 double anonymous_namespace{sqlitereader.cpp}::ToDouble (const char * *s*, const std::string & *m* = `T_gamalite_conversion_to_double_failed`)

Converts string to double.

Parameter *s* can not be `NULL` pointer. If conversion fails, exception is thrown.

**Parameters**

***s*** string to convert

***m*** error message if conversion fails

**Exceptions**

***GNU_gama::Exception::sqlitexc*** (p. ***11***)

Referenced by sqlite_db_readConfigurationInfo(), sqlite_db_readCoordinates(), sqlite_db_readCovarianceMatrix(), sqlite_db_readHeightDifferences(), sqlite_db_readObservations(), sqlite_db_readPoints(), and sqlite_db_readVectors().

#### 4.1.1.3 int anonymous_namespace{sqlitereader.cpp}::ToInteger (const char * *s*, const std::string & *m* = `T_gamalite_conversion_to_integer_failed`)

Converts string to integer.

Parameter *s* can not be `NULL` pointer. If conversion fails, exception is thrown.

**Parameters**

***s*** string to convert

***m*** error message if conversion fails

**Exceptions**

***GNU_gama::Exception::sqlitexc*** (p. ***11***)

Referenced by sqlite_db_readClusters(), sqlite_db_readCoordinates(), sqlite_db_readCovarianceMatrix(), sqlite_db_readObservations(), and sqlite_db_readVectors().

### 4.1.2 Variable Documentation

#### 4.1.2.1 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_configuration_not_found

**Initial value:**

```
"configuration not found"
```

error message, used in **GNU_gama::local::sqlite_db::SqliteReader::retrieve** (p. 11)

Referenced by GNU_gama::local::sqlite_db::SqliteReader::retrieve().

#### 4.1.2.2 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_conversion_to_double_failed

**Initial value:**

```
"conversion to double failed"
```

error message, used in conversion function when no better message can be used

#### 4.1.2.3 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_conversion_to_integer_failed

**Initial value:**

```
"conversion to integer failed"
```

error message, used in conversion function when no better message can be used

#### 4.1.2.4 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_database_not_open

**Initial value:**

```
"database not open"
```

error message, used in **GNU_gama::local::sqlite_db::SqliteReader::SqliteReader** (p. 10)

Referenced by GNU_gama::local::sqlite_db::SqliteReader::SqliteReader().

#### 4.1.2.5 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_invalid_column_value

**Initial value:**

```
"invalid column value"
```

error message, used in callbacks' to indicate bad value of database field

Referenced by sqlite_db_readClusters(), sqlite_db_readConfigurationInfo(), sqlite_db_readConfigurationText(), sqlite_db_readCoordinates(), sqlite_db_readCovarianceMatrix(), sqlite_db_readHeightDifferences(), sqlite_db_readObservations(), sqlite_db_readPoints(), and sqlite_db_readVectors().

#### 4.1.2.6 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_stand_point_cluster_with_multi_dir_sets

**Initial value:**

```
"StandPoint cluster with multiple directions sets"
```

error message, used in **sqlite_db_readObservations** (p. 17)

Referenced by sqlite_db_readObservations().

#### 4.1.2.7 const char* anonymous_namespace{sqlitereader.cpp}::T_gamalite_unknown_exception_in_callback

**Initial value:**

```
"unknown exception in SqliteReader's callback"
```

error message, used in callbacks' catch(...)

Referenced by sqlite_db_readClusters(), sqlite_db_readConfigurationInfo(), sqlite_db_readConfigurationText(), sqlite_db_readCoordinates(), sqlite_db_readCovarianceMatrix(), sqlite_db_readHeightDifferences(), sqlite_db_readObservations(), sqlite_db_readPoints(), and sqlite_db_readVectors().

## 4.2 GNU_gama Namespace Reference

### Namespaces

- namespace **Exception**
- namespace **local**

## 4.3 GNU_gama::Exception Namespace Reference

### Classes

- class **sqlitexc**

  ***Exception*** *(p. 5) class for* ***GNU_gama::local::sqlite_db::SqliteReader*** *(p. 9).*

## 4.4 GNU_gama::local Namespace Reference

**Namespaces**

- namespace **sqlite_db**

## 4.5 GNU_gama::local::sqlite_db Namespace Reference

**Classes**

- struct **ReaderData**

  ***GNU_gama::local::sqlite_db::SqliteReader*** *(p. 9) class private data*

- class **SqliteReader**

  *Reads LocalNetwork from SQLite 3 database.*

# 5 Class Documentation

## 5.1 GNU_gama::local::sqlite_db::ReaderData Struct Reference

**GNU_gama::local::sqlite_db::SqliteReader** (p. 9) class private data

**Public Member Functions**

- **ReaderData** ()

**Public Attributes**

- GNU_gama::local::LocalNetwork ∗ **lnet**
- std::string **algorithm**
- bool **correction_to_ellipsoid**
- double **latitude**
- GNU_gama::Ellipsoid **ellipsoid**
- GNU_gama::Exception::base ∗ **exception**
- sqlite3 ∗ **sqlite3Handle**
- std::string **configurationId**
- GNU_gama::local::StandPoint ∗ **currentStandPoint**
- GNU_gama::local::Vectors ∗ **currentVectors**
- GNU_gama::local::Coordinates ∗ **currentCoordinates**
- GNU_gama::local::HeightDifferences ∗ **currentHeightDifferences**
- GNU_gama::local::CovMat ∗ **currentCovarianceMatrix**

**Private Member Functions**

- **ReaderData** (const **ReaderData** &)
- **ReaderData** & **operator=** (const **ReaderData** &)

### 5.1.1 Detailed Description

**GNU_gama::local::sqlite_db::SqliteReader** (p. 9) class private data

**For internal use only.**

Contains all private data. In file **sqlitereader.h** (p. 18) is forward declaration of this struct. But declaration is only available in this file (translation unit). All members are public. Functions especially (`extern "C"`) callbacks can easy manipulate with this members. This is no OOP violation because we can think about functions in this file as **ReaderData** (p. 6) member functions. Functions outside this file can't access this structure because they know only forward declaration and **GNU_gama::local::sqlite_db::SqliteReader** (p. 9) has declared pointer to this struct private of course. However, there are some problems with callbacks visibility (see **SqliteReaderCallbackType** (p. 14) or **sqlite_db_readConfigurationInfo** (p. 15) for details).

Callbacks **sqlite_db_readObservations** (p. 17), ... and **sqlite_db_readCovarianceMatrix** (p. 17) have to share data between it's invocations. So they needs access to the same `StandPoint`, `Vectors`, etc. They also needs access to **exception** (p. 9). This is the reason why **ReaderData** (p. 6) contains pointer to `StandPoint` etc. Pointers **currentStandPoint** (p. 8), **currentVectors** (p. 8), **currentCoordinates** (p. 8), **currentHeightDifferences** (p. 8) and **currentCovarianceMatrix** (p. 8) temporary points to objects which are in use at the moment by the **exec()** (p. 2) caller and corresponding callback invocations. When processing of one object is finished, this pointer should by set to `NULL` pointer.

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 GNU_gama::local::sqlite_db::ReaderData::ReaderData () [inline]

It sets all member variables. Pointers are set to `NULL` pointers. Strings are initialised by `""` to satisfied GCC `-Weffc++` warning options.

References ellipsoid.

#### 5.1.2.2 GNU_gama::local::sqlite_db::ReaderData::ReaderData (const ReaderData &) [private]

disabled copy constructor

### 5.1.3 Member Function Documentation

#### 5.1.3.1 ReaderData& GNU_gama::local::sqlite_db::ReaderData::operator= (const ReaderData &) [private]

disabled assignment operator

### 5.1.4 Member Data Documentation

#### 5.1.4.1 std::string GNU_gama::local::sqlite_db::ReaderData::algorithm

Referenced by sqlite_db_readConfigurationInfo().

#### 5.1.4.2 std::string GNU_gama::local::sqlite_db::ReaderData::configurationId

configuration id in database

Referenced by GNU_gama::local::sqlite_db::SqliteReader::retrieve(), sqlite_db_readClusters(), and sqlite_db_readConfigurationInfo().

#### 5.1.4.3 bool GNU_gama::local::sqlite_db::ReaderData::correction_to_ellipsoid

Referenced by sqlite_db_readConfigurationInfo().

#### 5.1.4.4 GNU_gama::local::Coordinates* GNU_gama::local::sqlite_db::ReaderData::currentCoordinates

Referenced by sqlite_db_readClusters(), and sqlite_db_readCoordinates().

#### 5.1.4.5 GNU_gama::local::CovMat* GNU_gama::local::sqlite_db::ReaderData::currentCovarianceMatrix

provides access to same covariance matrix for callback readCovarianceMatrix and caller of **exec()** (p. 2) function

Referenced by sqlite_db_readClusters(), and sqlite_db_readCovarianceMatrix().

#### 5.1.4.6 GNU_gama::local::HeightDifferences* GNU_gama::local::sqlite_db::ReaderData::currentHeightDifferences

Referenced by sqlite_db_readClusters(), and sqlite_db_readHeightDifferences().

#### 5.1.4.7 GNU_gama::local::StandPoint* GNU_gama::local::sqlite_db::ReaderData::currentStandPoint

provides access to same stand point for callback readObservations and caller of **exec()** (p. 2) function

Referenced by sqlite_db_readClusters(), and sqlite_db_readObservations().

#### 5.1.4.8 GNU_gama::local::Vectors* GNU_gama::local::sqlite_db::ReaderData::currentVectors

Referenced by sqlite_db_readClusters(), and sqlite_db_readVectors().

#### 5.1.4.9 GNU_gama::Ellipsoid GNU_gama::local::sqlite_db::ReaderData::ellipsoid

Referenced by ReaderData(), and sqlite_db_readConfigurationInfo().

#### 5.1.4.10 GNU_gama::Exception::base* GNU_gama::local::sqlite_db::ReaderData::exception

an exception which was caught in callback or `NULL` if no exception was thrown

Referenced by sqlite_db_readClusters(), sqlite_db_readConfigurationInfo(), sqlite_db_readConfigurationText(), sqlite_db_readCoordinates(), sqlite_db_readCovarianceMatrix(), sqlite_db_readHeightDifferences(), sqlite_db_readObservations(), sqlite_db_readPoints(), sqlite_db_readVectors(), and GNU_gama::local::sqlite_db::SqliteReader::~SqliteReader().

#### 5.1.4.11 double GNU_gama::local::sqlite_db::ReaderData::latitude

Referenced by sqlite_db_readConfigurationInfo().

#### 5.1.4.12 GNU_gama::local::LocalNetwork* GNU_gama::local::sqlite_db::ReaderData::lnet

pointer to network object

Referenced by GNU_gama::local::sqlite_db::SqliteReader::retrieve(), sqlite_db_readClusters(), sqlite_db_readConfigurationInfo(), sqlite_db_readConfigurationText(), and sqlite_db_readPoints().

#### 5.1.4.13 sqlite3* GNU_gama::local::sqlite_db::ReaderData::sqlite3Handle

pointer to `struct sqlite3`

Referenced by GNU_gama::local::sqlite_db::SqliteReader::retrieve(), sqlite_db_readClusters(), GNU_gama::local::sqlite_db::SqliteReader::SqliteReader(), and GNU_gama::local::sqlite_db::SqliteReader::~SqliteReader().

The documentation for this struct was generated from the following file:

- **sqlitereader.cpp**

## 5.2 GNU_gama::local::sqlite_db::SqliteReader Class Reference

Reads LocalNetwork from SQLite 3 database.

`#include <sqlitereader.h>`

### Public Member Functions

- **SqliteReader** (const std::string &fileName)
- **~SqliteReader** ()
- void **retrieve** (LocalNetwork *&lnet, const std::string &configuration)

**Private Member Functions**

- **SqliteReader** (const **SqliteReader** &)
- **SqliteReader** & **operator=** (const **SqliteReader** &)

**Private Attributes**

- **ReaderData** ∗ **readerData**

### 5.2.1 Detailed Description

Reads LocalNetwork from SQLite 3 database.

### 5.2.2 Constructor & Destructor Documentation

#### 5.2.2.1 SqliteReader::SqliteReader (const std::string & *fileName*)

Opens a database connection.

**Parameters**

***fileName*** name of database file

References readerData, GNU_gama::local::sqlite_db::ReaderData::sqlite3Handle, and anonymous_-namespace{sqlitereader.cpp}::T_gamalite_database_not_open.

#### 5.2.2.2 SqliteReader::∼SqliteReader ()

Closes a database connection.

If function `sqlite3_close` returns another value then `SQLITE_OK` (there were some error), no action is performed.

References GNU_gama::local::sqlite_db::ReaderData::exception, readerData, and GNU_-gama::local::sqlite_db::ReaderData::sqlite3Handle.

#### 5.2.2.3 GNU_gama::local::sqlite_db::SqliteReader::SqliteReader (const SqliteReader &) [private]

disabled copy constructor

### 5.2.3 Member Function Documentation

#### 5.2.3.1 SqliteReader& GNU_gama::local::sqlite_db::SqliteReader::operator= (const SqliteReader &) [private]

disabled assignment operator

#### 5.2.3.2 void SqliteReader::retrieve (LocalNetwork *& *lnet*, const std::string & *configuration*)

Reads configuration *configuration* from database.

If *lnet* is a `NULL` pointer, new `LocalNetwork` is created. Type of network depends on algorithm fetched from database.

**Exceptions**

***GNU_gama::Exception::sqlitexc*** (p. ***11***)

References GNU_gama::local::sqlite_db::ReaderData::configurationId, anonymous_namespace{sqlitereader.cpp}::exec(), GNU_gama::local::sqlite_db::ReaderData::lnet, readerData, GNU_gama::local::sqlite_db::ReaderData::sqlite3Handle, sqlite_db_readClusters(), sqlite_db_readConfigurationInfo(), sqlite_db_readConfigurationText(), sqlite_db_readPoints(), and anonymous_namespace{sqlitereader.cpp}::T_gamalite_configuration_not_found.

### 5.2.4 Member Data Documentation

#### 5.2.4.1 ReaderData* GNU_gama::local::sqlite_db::SqliteReader::readerData [private]

pointer to private data

Referenced by retrieve(), SqliteReader(), and ~SqliteReader().

The documentation for this class was generated from the following files:

- **sqlitereader.h**
- **sqlitereader.cpp**

## 5.3 GNU_gama::Exception::sqlitexc Class Reference

**Exception** (p. 5) class for **GNU_gama::local::sqlite_db::SqliteReader** (p. 9).

`#include <sqlitereader.h>`

**Public Member Functions**

- **sqlitexc** (const std::string &message)
- virtual **sqlitexc** * **clone** () const
- virtual void **raise** () const

### 5.3.1 Detailed Description

**Exception** (p. 5) class for **GNU_gama::local::sqlite_db::SqliteReader** (p. 9).

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 GNU_gama::Exception::sqlitexc::sqlitexc (const std::string & *message*) [inline]

**Parameters**

***message*** sqlite database error message or SqliteReader message

Referenced by clone().

### 5.3.3 Member Function Documentation

#### 5.3.3.1 virtual sqlitexc* GNU_gama::Exception::sqlitexc::clone () const [inline, virtual]

Clones an exception.

**For internal use only.**

The way as it is used in callback functions:

```
// ... try block
catch (GNU_gama::Exception::base& e)
    {
        d->exception = e.clone();
    }
return 1;
```

References sqlitexc().

#### 5.3.3.2 virtual void GNU_gama::Exception::sqlitexc::raise () const [inline, virtual]

Rethrows an exception polymorphically.

**For internal use only.**

The way as it is used in function `exec` in file **sqlitereader.cpp** (p. 12):

```
if (readerData->exception != 0)
    {
        readerData->exception->raise();
    }
```

The documentation for this class was generated from the following file:

- **sqlitereader.h**

# 6 File Documentation

## 6.1 sqlitereader.cpp File Reference

Implementation of **GNU_gama::local::sqlite_db::SqliteReader** (p. 9).

# C Použitá nastavení kompilátoru *GCC*

Zde jsou uvedeny parametry určující, v jakých chvílích bude kompilátor *GCC* hlásit varování. Parametry uvedené v následujícím seznamu byly použity při vývoji třídy `SqliteReader`.

```
-ansi
-Wall
-pedantic
-Wextra
-Weffc++
-Wconversion
-Wsign-conversion
-Wfloat-equal
-Wno-div-by-zero
-Wmissing-declarations
-Wlogical-op
-Wabi
-Wold-style-cast
-Woverloaded-virtual
-Wshadow
-Wcast-qual
-Wwrite-strings
-Wredundant-decls
-fno-nonansi-builtins
-Wctor-dtor-privacy
```