SimpleDB

Vaším úkolem je vytvořit jednoduchou knihovnu, která bude sloužit jako jednoduchý (ale obecný) databázový systém – obdoba relačních databází. Pro Vaši databázi je dále potřeba vytvořit demonstrační příklad obsahující jednoduchou kartotéku dat, která bude obsahovat alespoň 2 tabulky, které budou vzájemně propojeny pomocí některého pole.

Výsledkem semestrální práce jsou dva propojené projekty – projekt databáze (DLL knihovna) a demonstrační aplikace (spustitelný konzolový program).

SimpleDB

SimpleDB představuje knihovnu, která nabízí velmi triviální tabulkovou databázi bez složitého dotazovacího jazyka.

Základní vlastnosti:

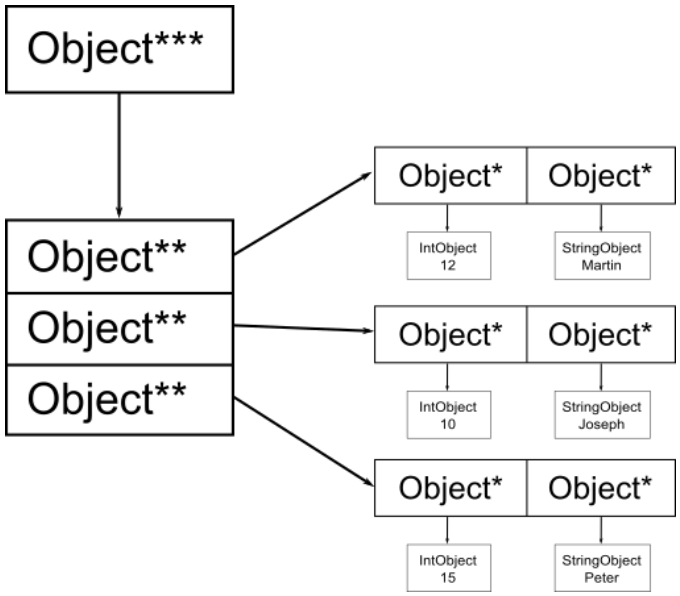
- Databáze je identifikována svým **jménem**.
- Pro ukládání dat využívá tabulky.
  - Tabulka je identifikována svým **jménem**.
  - Tabulka se skládá z **několika datových polí (sloupečků)**.
  - Tabulka může obsahovat 0, 1 nebo více řádků dat.
- Datové pole je jednoho ze tří typů: **int**, **double**, **string** (bez omezení délky řetězce)
  - Pole musí vždy nabývat platné hodnoty, základní API neuvažuje koncept „*null*“.
  - Pokud bude jako backend sloužit binární soubor je možné považovat string jako řetězec s pevně definovanou maximální délkou. Je nutné rozšířit API knihovny pro umožnění definice maximální délky a následně korektně kontrolovat plnění této podmínky.
- Pro uložení struktury tabulek i jejich dat **využijte soubory**.
  - Je možné použít **textové i binární soubory** (i v kombinaci).
  - Je možné (a doporučené) rozdělit schéma a data v tabulkách do samostatných souborů. Příklady názvů datových souborů:
    - NazevDatabaze_NazevTabulky.schema
    - NazevDatabaze-NazevTabulky-schema.dat
    - NazevDatabazeXNazevTabulkyXs
  - Formát použitý pro zápis dat a struktury v souborech není pevně stanoven – navrhněte jej.
- Databázové operace:
  - **select (iterátor)** – umožňuje projít veškerá data ve vybrané tabulce, data jsou měnitelná, select lze využít jako update
  - **insert()** – vložení nového řádku do tabulky
  - **remove()** – smazání vybraného řádku (dle rowid) z tabulky
  - **commit()** – uložení stavu tabulky z paměti do datových souborů
  - API hojně využívá ukazatele – jejich dynamická alokace a dealokace je řešena kompletně knihovnou! Uživatel databáze nemusí řešit dealokování žádných objektů. Vizte ukázkový příklad použití knihovny.
  - Několik metod je označeno jako bonusové – jejich implementace a použití není povinné. Jejich realizace bude poznamenána a poslouží jako případný plus bod u zkoušky.
- Neuvažuje se:
  - Indexy, SQL nebo jiný dotazovací jazyk.

- Databáze realizována jako **DLL knihovna**.
    - V zásadě lze celou dobu vyvíjet demonstrační aplikaci a DB v jednom projektu a rozdělit to až na konci. Pokud budou důsledně odděleny funkce z DB knihovny a aplikace, rozdělení do dvou projektů zabere maximálně 15 minut.

Doporučená reprezentace dat v paměti:

- Jednotlivé datové hodnoty v tabulce (a také popis schématu db) je realizováno pomocí tříd *IntObject*, *StringObject*, *DoubleObject* a *FieldObject*; ty jsou potomky třídy *Object* a využívají polymorfizmu k reprezentaci celé tabulky.

- Data tabulek je možné reprezentovat jako *Object**** (2D matice ukazatelů na *Object*)



Tabulka: users	
id	**Name**
12	Martin
10	Joseph
15	Peter

- Struktura tabulek jako *FieldObject*** (pole ukazatelů na *FieldObject* – jednotlivé sloupce)

Předepsané API (soubor dbapi.h)

```
// Typ datového pole
enum struct FieldType {
	Integer,
	Double,
	String,
	Field
};

// Databáze
Class DLL_SPEC Db {

	// Otevře databázi
	static Db* open(std::string database);
	// Uzavře databázi (dealokuje paměťové prostředky)
	void close();

	// Vytvoří novou tabulku
	Table* createTable(std::string name, int fieldsCount, FieldObject** fields);
	// Otevře existující tabulku
	Table* openTable(std::string name);
	// Otevře tabulku (pokud neexistuje, vytvoří automaticky novou)
	Table* openOrCreateTable(std::string name, int fieldsCount, FieldObject** fields);

	// Alokuje objekt „int“
	static Object* Int(int value);
	// Alokuje objekt „double“
	static Object* Double(double value);
	// Alokuje objekt „string“
	static Object* String(std::string value);
	// Alokuje objekt „field“
	static FieldObject* Field(std::string name, FieldType type);
}
// ------------------------------------------------------

// Rozhraní definující podmínku – pouze pro bonusové metody
class DLL_SPEC Condition {
	virtual ~Condition() { }
	virtual bool matches(int fieldCount, FieldObject** fields, Object** row) const = 0;
};
// ------------------------------------------------------

// Rozhraní iterátoru (select)
class DLL_SPEC Iterator {
public:
	virtual ~Iterator();

	// Posun na další řádek (vrací true, pokud je iterátor platný; logika podle Java Iterator)
	virtual bool moveNext() = 0;
	// Vrací pole Object* obsahující data řádku
	virtual Object** getRow() = 0;
	// Vrací interní rowId aktuálního řádku
	virtual int getRowId() = 0;
	// Uzavře iterátor (dealokuje paměťové prostředky)
	virtual void close() = 0;
};
```

```
// Tabulka
class DLL_SPEC Table {
public:
	// Vložení nového řádku do tabulky (pole Object* (pro jednotlivé hodnoty sloupečků))
	void insert(Object** row);
	// Smazání vyrabného řádku z tabulky
	void remove(int rowid);

	// Select – vytvoří iterátor k procházení tabulky
	Iterator* select();

	// Commit – přenese změny z paměti do datových souborů
	void commit();

	// Uzavře tabulku (dealokuje paměťové prostředky)
	void close();

	// Vrací počet řádků v tabulce
	int getRowCount() const;

	// Vrací pole FieldObject* popisující sloupečky tabulky
	FieldObject** getFields() const;

	// Vrací počet sloupečků
	int getFieldCount() const;


	// ============== Bonusové metody: ================
	// Select s podmínkou
	Iterator* select(Condition* condition) { throw 0; }
	// Nalezení rowId s podmínkou
	int findRowId(Condition* condition) { throw 0; }
	// Update – aktualizuje řádky vyhovující podmínce, aktualizaci provádí funkce „callback“
	// callback na vstupu obdrží data řádku a vrací data
	void update(Condition* condition, std::function<void(Object**)> callback) { throw 0; }
}



// Polymorfní datový objekt (reprezentuje jednu datovou hodnotu v tabulce)
// Rozhraní vyhovuje základním typům int, double, string; pro typ „field“ je rozhraní rozšířeno
class DLL_SPEC Object {
public:
	Object();
	virtual ~Object();

	// Gettery a settery podle typu
	// Jejich funkce je definována jen v případě, že aktuální objekt je odpovídajícího typu
	// Automatické konverze v základním API nejsou vyžadovány

	virtual std::string getString() const;
	virtual void setString(std::string value);

	virtual int getInt() const;
	virtual void setInt(int value);

	virtual double getDouble() const;
	virtual void setDouble(double value);

	// Vrací true, pokud aktuální objekt představuje daný typ
	virtual bool isType(FieldType type) const;
};
```

```
class DLL_SPEC IntObject : public Object {
public:
	IntObject() : value(0) {}
	IntObject(int v) : value(v) {}
…
};

class DLL_SPEC DoubleObject : public Object {
public:
	DoubleObject() : value(0.0) {}
	DoubleObject(double v) : value(v) {}
…
};

class DLL_SPEC StringObject : public Object {
public:
	StringObject() : value("") {}
	StringObject(std::string v) : value(v) {}
…
};

// Objekt popisující sloupeček „field“
class DLL_SPEC FieldObject : public Object {
public:
	FieldObject() {}
	FieldObject(std::string name, FieldType type) :name(name), type(type) {}

	virtual bool isType(FieldType type) const override;

	// Název sloupečku
	std::string getName() const { return name; }
	// Typ sloupečku
	FieldType getType() const { return type; }
};
```

Ukázka použití (soubor example.cpp):

```
#include <db.h>

…

// Vytvoření/otevření db
Db* db = Db::open("testdb");

// Vytvoření/otevření tabulky
auto idField = Db::Field("id", FieldType::Integer);
auto nameField = Db::Field("name", FieldType::String);
auto userFields = combineToDefinition(idField, nameField);
Table* users = db->openOrCreateTable("users", 2, userFields);

// Vložení řádku do tabulky
auto id = Db::Int(15);
auto name = Db::String("Peter");
auto row = combineToRow(id, name);
users->insert(row);

// Select
auto it = users->select();
while (it->moveNext())
{
	auto row = it->getRow();
	cout << row[0]->getInt() << ": " << row[1]->getString() << endl;
}
it->close();

// Uložení tabulky na disk
users->commit();

// Uzavření tabulky (a dealokace paměťových prostředků)
users->close();

// Uzavření db (a dealokace paměťových prostředků)
db->close();
```

Pomocné funkce (vizte soubor helpful.h a ukázku použití):

```
template<typename A>
Object** combineToRow(A a) {
	return new Object*[1]{ a };
}

template<typename A, typename B>
Object** combineToRow(A a, B b) {
	return new Object*[2]{ a, b };
}

template<typename A, typename B, typename C>
Object** combineToRow(A a, B b, C c) {
	return new Object*[3]{ a, b, c };
}

template<typename A, typename B, typename C, typename D>
Object** combineToRow(A a, B b, C c, D d) {
	return new Object*[4]{ a, b, c, d };
}

template<typename A, typename B, typename C, typename D, typename E>
Object** combineToRow(A a, B b, C c, D d, E e) {
	return new Object*[5]{ a, b, c, d, e };
}


template<typename A>
FieldObject** combineToDefinition(A a) {
	return new FieldObject*[1]{ a };
}

template<typename A, typename B>
FieldObject** combineToDefinition(A a, B b) {
	return new FieldObject*[2]{ a, b };
}

template<typename A, typename B, typename C>
FieldObject** combineToDefinition(A a, B b, C c) {
	return new FieldObject*[3]{ a, b, c };
}

template<typename A, typename B, typename C, typename D>
FieldObject** combineToDefinition(A a, B b, C c, D d) {
	return new FieldObject*[4]{ a, b, c, d };
}

template<typename A, typename B, typename C, typename D, typename E>
FieldObject** combineToDefinition(A a, B b, C c, D d, E e) {
	return new FieldObject*[5]{ a, b, c, d, e };
}
```

Demonstrační aplikace

Demonstrační aplikace bude využívat vytvořenou databázovou knihovnu a bude demonstrovat její funkcionality.

- Je nutné využít vytvořenou databázovou knihovnu.
- Téma demonstrační aplikace je libovolné.
- Minimálně je třeba využívat 2 tabulky spojené „relací“.
- Aplikace musí nabídnout základní CRUD operace.

- **Odevzdaná semestrální práce musí obsahovat databázi s ukázkovými daty pro jednodušší otestování funkčnosti aplikace (tj. minimálně 5 záznamů pro každou tabulku).**

Technické požadavky na realizaci semestrální práce

- **Připravené rozhraní je nutné zcela dodržet**. Návrh a implementace chybějících částí je zcela na Vás. Detaily vizte přiložené soubory.

- Veškerá dynamická paměť musí být **korektně dealokována**. Není přípustné nechat paměť alokovanou déle, než je to nutné.

- Vytvoření DLL knihovny je možné shlédnout na:
  - https://github.com/MicrosoftDocs/cpp-docs/blob/master/docs/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp.md
  - https://docs.microsoft.com/cs-cz/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=vs-2019

- DLL poznámky:
  - Projekt SimpleDB musí být nastaven, že jeho výstupem je DLL knihovna.
    - C++ - preprocesor – definice: WIN32 / WIN64, _WINDLL
  - Projekt DemonstračníAplikace musí mít nastaveno:
    - C++ - preprocesor – definice: WIN32 / WIN64
    - C++ - includes – relativní cesta k H souborům knihovny (/I)
    - Linker – knihovna – název knihovny SimpleDB.lib
    - Linker – cesty ke knihovnám – relativní cesta k SimpleDB.lib (/LIBPATH)
    - Pokud nebudou v nastavení projektu použity relativní cesty – nepůjde projekt na jiném pc zkompilovat a semestrální práce nebude akceptována.
  - Příklad konfigurace solution a projektů je přiložen k zadání semestrální práce.

- **Není dovoleno** využívat **globální a statické proměnné**.

- **Program vypracovaný pro platformu Windows musí obsahovat projektové soubory pro Visual Studio 2017 nebo novější, pomocí kterých lze program přeložit.**

  Pro unixové platformy musí být přiložen makefile. Zdrojové kódy, které není možné přeložit (chybějící projektové/zdrojové soubory, syntaktické chyby v kódu), nebudou akceptovány.

  Před odevzdáním musí být projekt uklizen od veškerých binárních souborů (spustitelných, tmp, cache, objektových – pomocí Build – Clean solution | make clean), dále neodevzdávejte složky: „debug“, „release“, „output“, „x86“, „x64“, „.vs“. Vyčištěné zdrojové kódy včetně souborů projektu odevzdávejte ve formátu ZIP.

- **Vyžaduje se platný kód dle standardu INCITS/ISO/IEC 14882:2017 („C++ 17“). Nejsou dovolena žádná proprietární rozšíření jednotlivých kompilátorů.**

Nejzazší termín odevzdání semestrální práce je 31. 12. 2019 23:59.

Semestrální práce musí být vypracována samostatně, není přípustná žádná shoda s jinou prací. Student musí být schopen vytvořenou práci okomentovat a vysvětlit při ústní obhajobě.