An example of applying variadic template to C code

•••

Masako Toda (mtoda@blizzard.com)
Blizzard Entertainment

What to expect

- Variadic templates: Templates that take a variable number of arguments.
- Nice! But when/where can I use it? Ah, C style SQLite code!
- Mr. Kenny Kerr's "SQLite with Modern C++" covers a lot more than my talk.
- One more trick, thanks to my boss, Ben Deane.

Problem

There is a SQLite database that consists of two tables:

[Students] table

- Name : text
- Age : integer

[Teachers] table

- Name : text
- Subject : text
- Salary : integer

I need to write functions to insert records into these tables.

C style solution

```
bool InsertStudent(sqlite3* db, const string& name, int age)
{
    if (!db) return false;     ←----------------------------------------------- Check if database is open

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,    ←----------------------------------------------- Call prepare
        "INSERT INTO students (name, age) VALUES (?,?)",
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    if (sqlite3_bind_text(stmt, ++index,    ←---------------------------------- Call bind for the 1st parameter
        name.c_str(), name.length(), SQLITE_STATIC) != SQLITE_OK)
        return false;

    if (sqlite3_bind_int(stmt, ++index, age) != SQLITE_OK)    ←--------------- Call another bind for the 2nd parameter
        return false;

    return (sqlite3_step(stmt) != SQLITE_DONE);    ←------------------------- Call step
}
```

C style solution

```
bool InsertStudent(sqlite3* db, const string&
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO students (name, age) VAL
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    if (sqlite3_bind_text(stmt, ++index,
        name.c_str(), name.length(), SQLITE_S
        return false;

    if (sqlite3_bind_int(stmt, ++index, age)
        return false;

    return (sqlite3_step(stmt) != SQLITE_DONE
}
```

```
bool InsertTeacher(sqlite3* db, const string& name,
    const string& subject, int salary)
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO teachers (name, subject, salary) VALUES (?,?,?)",
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    if (sqlite3_bind_text(stmt, ++index,
        name.c_str(), name.length(), SQLITE_STATIC) != SQLITE_OK)
        return false;

    if (sqlite3_bind_text(stmt, ++index,
        subject.c_str(), subject.length(), SQLITE_STATIC) != SQLITE_OK)
        return false;

    if (sqlite3_bind_int(stmt, ++index, salary) != SQLITE_OK)
        return false;

    return (sqlite3_step(stmt) != SQLITE_DONE);
}
```

bind overload

```
bool InsertStudent(sqlite3* db, const string&
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO students (name, age) VAL
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    try {
        bind(stmt, ++index, name);
        bind(stmt, ++index, age);
    } catch (...) { return false; }

    return (sqlite3_step(stmt) != SQLITE_DONE
}
```

```
void bind(sqlite3_stmt* stmt, int index, int i);
void bind(sqlite3_stmt* stmt, int index, const string& str);
```

```
bool InsertTeacher(sqlite3* db, const string& name,
    const string& subject, int salary)
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO teachers (name, subject, salary) VALUES (?,?,?)",
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    try {
        bind(stmt, ++index, name);
        bind(stmt, ++index, subject);
        bind(stmt, ++index, salary);
    } catch (...) { return false; }

    return (sqlite3_step(stmt) != SQLITE_DONE);
}
```

Focus on bind

```
bool InsertStudent(sqlite3* db, const string&
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO students (name, age) VAL
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    try {
        bind(stmt, ++index, name);
        bind(stmt, ++index, age);
    } catch (...) { return false; }

    return (sqlite3_step(stmt) != SQLITE_DONE
}
```

```
bool InsertTeacher(sqlite3* db, const string& name,
    const string& subject, int salary)
{
    if (!db) return false;

    sqlite3_stmt* stmt = nullptr;

    if (sqlite3_prepare(db,
        "INSERT INTO teachers (name, subject, salary) VALUES (?,?,?)",
        -1, &stmt, 0) != SQLITE_OK)
        return false;

    int index = 0;

    try {
        bind(stmt, ++index, name);
        bind(stmt, ++index, subject);
        bind(stmt, ++index, salary);
    } catch (...) { return false; }

    return (sqlite3_step(stmt) != SQLITE_DONE);
}
```

BindAll function

```
try { BindAll(stmt, name, subject, salary); } catch (...) { return false; }
```

```
try { BindAll(stmt, name, age); } catch (...) { return false; }
```

```
int index = 0;

try {
    bind(stmt, ++index, name);
    bind(stmt, ++index, age);
} catch (...) { return false; }
```

```
int index = 0;

try {
    bind(stmt, ++index, name);
    bind(stmt, ++index, subject);
    bind(stmt, ++index, salary);
} catch (...) { return false; }
```

variadic template - recursive expansion

```
template <typename ... Ts> struct Binder;      ←-------------------------------- variadic template class - forward declaration

template <> struct Binder<>                    ←-------------------------------- Termination
{
    static void Bind(sqlite3_stmt* stmt, int& index) {}
};

template <typename T, typename ... Ts> struct Binder<T, Ts...>    ←------------- Expansion
{
    static void Bind(sqlite3_stmt* stmt, int& index, T arg, Ts ... args)
    {
        bind(stmt, ++index, arg);                        ←---------------------- Bind for the 1st argument
        Binder<Ts...>::Bind(stmt, index, args...);       ←---------------------- Expand template recursively for the rest
    }
};

template <typename ... Ts> void BindAll(sqlite3_stmt* stmt, Ts ... args)    ←--- variadic template function
{
    int index = 0;
    Binder<Ts...>::Bind(stmt, index, args...);
}
```

parameter pack expansion

```
template <typename ... Ts> struct Binder;

template <> struct Binder<>
{
    static void Bind(sqlite3_stmt* stmt, int&
};

template <typename T, typename ... Ts> struct
{
    static void Bind(sqlite3_stmt* stmt, int&
    {
        bind(stmt, ++index, arg);
        Binder<Ts...>::Bind(stmt, index, args
    }
};

template <typename ... Ts> void BindAll(sqlit
{
    int index = 0;
    Binder<Ts...>::Bind(stmt, index, args...)
}
```

Calls bind function for each of args and returns a list of 0.

```
template <typename ... Ts> void BindAll(sqlite3_stmt* stmt, const Ts& ... args)
{
    int index = 0;
    (void) std::initializer_list<int> {(bind(stmt, ++index, args), 0)... };
}
```

fold expression (C++ 17)

recursive expansion (bad)

```
template <typename ... Ts> struct Binder;

template <> struct Binder<>
{
    static void Bind(sqlite3_stmt* stmt, int&
};

template <typename T, typename ... Ts> struct
{
    static void Bind(sqlite3_stmt* stmt, int&
    {
        bind(stmt, ++index, arg);
        Binder<Ts...>::Bind(stmt, index, args
    }
};

template <typename ... Ts> void BindAll(sqlit
{
    int index = 0;
    Binder<Ts...>::Bind(stmt, index, args...)
}
```

parameter pack expansion (good)

```
template <typename ... Ts> void BindAll(sqlite3_stmt* stmt, const Ts& ... args)
{
    int index = 0;
    (void) std::initializer_list<int> {(bind(stmt, ++index, args), 0)... };
}
```

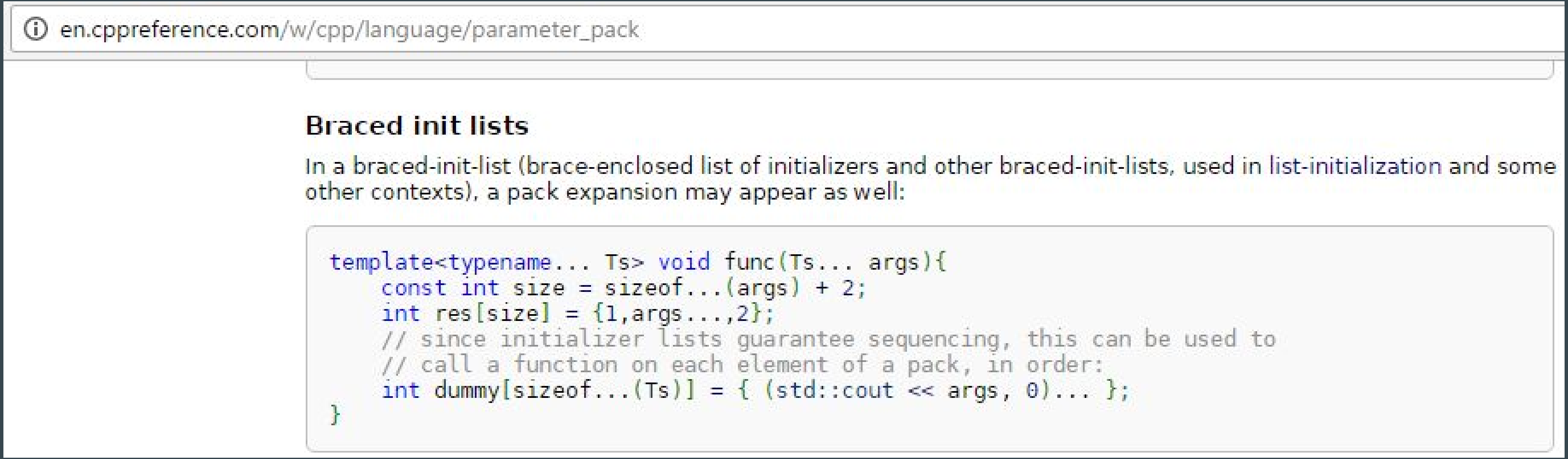
fold expression (i like it!)

```
template <typename ... Ts> void BindAll(sqlite3_stmt* stmt, const Ts& ... args)
{
    int index = 0;
    (bind(stmt, ++index, args), ...);
}
```

Thank you for your time!

- References: http://cppreference.com, “SQLite with Modern C++” https://www.youtube.com/watch?v=YqZaeM6iPwE
- Source code: https://github.com/masakotoda/cppcon16/
- Thanks to: Mr. Roland Bock, Ben Deane, and my dear colleagues ☺

Appendix - in cppreference


en.cppreference.com/w/cpp/language/parameter_pack

Braced init lists

In a braced-init-list (brace-enclosed list of initializers and other braced-init-lists, used in list-initialization and some other contexts), a pack expansion may appear as well:

```cpp
template<typename... Ts> void func(Ts... args){
    const int size = sizeof...(args) + 2;
    int res[size] = {1,args...,2};
    // since initializer lists guarantee sequencing, this can be used to
    // call a function on each element of a pack, in order:
    int dummy[sizeof...(Ts)] = { (std::cout << args, 0)... };
}
```