C++ in LoXiM

- Marek Dopiera

Motto

„A humble thing but perfectly done is noble.”

Sir Royce

Agenda

- Motivation
- The cause of poor quality of code in LoXiM
- Do we need C++ features?
- Theoretical introduction to smart pointers, RAII, const correctness and exceptions
- Examples of poor code
- Error summary and classification
- Presentation of some new, hopefully better written code

Motivation

- We all know the slogans which are about to appear, but ...
- Practice has prooven us wrong
- Expressing the slogans in a programming language is sometimes tricky
- Probable discussion on the suggested solutions will certainly be worthsome
- Much of the content is only my opinion, so it is natural that you may not agree with it

The cause of poor quality of code in LoXiM (examples)

- Poor C++ knowledge and an attempt to write code like in C or Java
- Assumption, that I know better what is good or bad than C++ creators
- Unverified gossips:
  - Exceptions are extremely slow
  - References are just syntactic sugar, which make the code harder to understand
  - Constructors shouldn't throw and should be light
  - `char*` is faster and fancier than std::string
- Not using constructs which make the code more readable
- Not knowing the C++ semantics

Do we need exceptions?

- Actually, can we avoid them?

```
$ cat no-exceptions.cpp
int main()
{
                char *a = new char[10000000000];
}
g++ -fno-exceptions -o no-exceptions no-exceptions.cpp
$ ./no-exceptions
terminate called after throwing an instance of 'std::bad_alloc'
  what():  St9bad_alloc
```

  - No!
- What about performance?
- What about the existing code base?
- Nevertheless, I say YES

Exceptions, example of readability

```
int DescriptorInputStream::read(char *buffer, unsigned long int off, unsigned long int
length, sigset_t *sigmask, int *cancel)
{
        fd_set rfds;
        int res;
        sigset_t old_mask;
        if (status < 0 && status != IS_STATUS_TIMEDOUT)
                return status;

        FD_ZERO(&rfds);
        FD_SET(fd, &rfds);
        pthread_sigmask(SIG_SETMASK, sigmask, &old_mask);
        if (*cancel){
                status = IS_STATUS_CANCEL;
                return status;
        }
        res = pselect(fd+1, &rfds, NULL, NULL, NULL, &old_mask);
        pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
        if (res == 1){
                return read(buffer, off, length);

        } else {
                if (res == 0)
                        status = IS_STATUS_TIMEDOUT;
                else
                        if (errno == EINTR && *cancel)
                                status = IS_STATUS_CANCEL;
                        else
                                status = IS_STATUS_OTHERERROR;
                return status;
        }
```

Exceptions, example of readability

```
void FileDataStream::read(char *buf, size_t len, const sigset_t &mask,
                const bool &cancel)
{
        while (len){
                fd_set fds;
                FD_ZERO(&fds);
                FD_SET(fd, &fds);
                int res;
                {
                        Masker m(mask);
                        if (cancel)
                                throw OperationCancelled();
                        res = pselect(fd+1, &fds, NULL, NULL, NULL,
                                        &m.get_old_mask());
                }
                if (res < 0)
                        throw ReadError(errno);
                if (res != 1){
                        //the OS is cheating ;)
                        throw ReadError(EIO);
                }
                res = ::read(fd, buf, len);
                if (res < 0)
                        throw ReadError(errno);
                if (res == 0)
                        throw ReadError(ENODATA);
                len -= res;
        }
}
```

Do we need C++ constructs?

- References: YES
- Objects: YES
- Templates: YES
- Answer: YES

Theoretical introduction

- Exceptions
- C++, contrary to Java, strictly specifies objects' lifetime
- Smart pointers
- The former allow us to program elegant RAII and exception safety
- Const correctness
- References
- Templates

Examples of poor code

- The reason is not to laugh at people, but to learn by example
- The samples are collected from almoast every part of LoXiM

```
bool LoximServer::LoximSession::authorize(const char *login, const char *passwd)
{
        QueryResult *qres;
        int res;
        bool correct;
        res = execute_statement("begin", &qres);
        if (res)
                return false;
        delete qres;
        res = execute_statement(("validate " + string(login) + " " +
string(passwd)).c_str(), &qres);
        if (res){
                return false;
        }
        //do the check
        if (qres->type() != QueryResult::QBOOL){
                delete qres;
                return false;
        } else {
                correct = ((QueryBoolResult*)qres)->getValue();
                delete qres;
        }
        res = execute_statement("end", &qres);
        if (res)
                return false;
        delete qres;

        if (correct) {
                stats->setUserLogin(string(login));
        }
        return correct;
}
```

Problems

- Manual static cast
  - If the instance of QueryResult was a subject of multiple inheritance, it would probably cause a SEGFAULT
  - dynamic_cast should be used
- Arguments should be const string&
- Redundant explicit conversion to string

```
int LoximServer::prepare()
{
        socket = new
TCPIPServerSocket(const_cast<char*>(hostname.c_str()), port);
        int res;
        if ((res = socket->bind()) == SOCKET_CONNECTION_STATUS_OK){
                prepared = 1;
                return 0;
        } else
                return res;
}
```

Problems

- What is the reason for the existance of this method? To comply with RAII, it should be a part of the ctor
- The cons_cast is ugly, but the protocol's API enforces us to do it

```
int LogIO::readString( int fileDes, string &str )
{
        int errCode;
        unsigned int n = 0;
        int len;
        char *buffer;

        if( ( errCode = readInt( fileDes, len ) ) ) return errCode;

        buffer = new char[len];

        while( n < (unsigned) len )
        {
                int size = ::read( fileDes, buffer+n, len-n );
                if( !size ) return UNEXPECTED_EOF_ERROR;
                n += size;
                if( size == -1 ) return errno;
        }

        str.clear();
        str.append(buffer, len);
        delete[] buffer;

        return 0;
}
```

Problems

- There is a memory leak if read returns an error
- Conclusion: writing methods with many execution paths having manual memory management is error prone
- Transfering the responsibility of memory management to the compiler by using smart pointers solves the problem

```
LockManager::LockManager() :
err(ErrorConsole::get_instance(EC_LOCK_MANAGER))
{
        transaction_locks = new TransactionIdMap;
        map_of_locks      = new DBPhysicalIdMap;
        single_lock_id    = 0;
        mutex = new Mutex();
        mutex->init();
}
```

Problems

- Why aren't the subobjects contained in LockManager?
- Memory leak will appear, if the second allocation fails
- Smart pointers are enough to avoid that
- Why does the mutex have the init() method? It should be inlined in the constructor.
- Initialization lists should be used.

```
//From Store::Buffer
PagePointer* getPagePointer(TransactionID* tid, unsigned short
fileID, unsigned int pageID);
```

Problems

- It is not clear whether the caller will become the owner of the returned object or not. In this case he will.
- If auto_ptr was used, hardly anybody would have any doubts
- It is not clear how long should tid live. In this case this is only during the execution of this method
- If const auto_ptr& or a reference were used, it would have been clear

```
//From QueryParser::TreeNode
virtual string getCard(){return this->card;}
```

Problems

- virtual inline functions seldom are a good trade-off
- This class is always used polymorphically, so the compiler can't inline the function, however it has to increase the executable's size
- It is probably much more desireble to return a const reference to the string instead of the value; depending on the use, it might save creating a temporary object (in current implementation it will be created, because the function will not be inlined)
- This method is not a mutator, so it should have a const keyword

```
const int Package::serialize(char **data)
{
        PackageBufferWriter *writer=new PackageBufferWriter();
        serializeW(writer);
        int size=writer->getSize()+5;
        *data=(char*)malloc(size);
        if(!(*data))
        {
                delete writer;
                return -1;
        };
        memcpy(*data,writer->getPackageBuffer(getPackageType()),size);
        delete writer;
        return size;
}
```

Problems

- Returning a const built-in type by value doesn't make any sense, the const keyword should be removed
- The object writer should not be allocated on heap, stack is definitely big enough
- -1 is hardcoded, the user will not have any clue what it means

```
ClassGraph::~ClassGraph() {
        vector<LogicalID*> lidsToDel;
        for(MapOfClassVertices::iterator i = classGraph.begin(); i !=
classGraph.end(); i++) {
                lidsToDel.push_back((*i).first);
                delete (*i).second;
        }
        for(MapOfInvariantVertices::iterator i = invariants.begin(); i !=
invariants.end(); i++) {
                delete (*i).second;
        }
        for(vector<LogicalID*>::iterator i = lidsToDel.begin(); i !=
        lidsToDel.end(); ++i) {
                delete (*i);
        }
}
```

Problems

- This is an example of an overblown destructor
- This destructor might throw (on push_back operation), which is unacceptable

```
//From QueryExecutor::QueryResult
virtual bool equal(QueryResult *r)=0;
```

Problems

- The method should be marked as const
- It should not require write access to the argument (const keyword should be added)
- It should use a reference rather than a pointer; comparing with a NULL doesn't make sense anyway

```
int LogManager::syncLog(unsigned int toLSN)
{
        if (sync_file_range(fd, 0, toLSN, SYNC_FILE_RANGE_WAIT_BEFORE |
SYNC_FILE_RANGE_WRITE | SYNC_FILE_RANGE_WAIT_AFTER) < 0)
                return ESyncLog;
#ifdef DEBUG_MODE
        debug_printf(*ec, "LogManager::syncLog up to %d done.\n",
toLSN);
#endif
        return 0;
}
```

Problems

- sync_file_range is Linux-specific

```
char *Client::ClientConsole::line_provider(const char *prompt)
{
        char *line = 0;
        size_t len;
        if (file){
                if ((getline(&line, &len, file) == -1) || !line){
                        fclose(file);
                        file = 0;
                        return line_provider(prompt);
                } else {
                        if (feof(file)){
                                fclose(file);
                                file = 0;
                        }
                        len = strlen(line);
                        if (len > 0 && line[len - 1] == '\n')
                                line[len - 1] = 0;
                        return line;
                }
        }
        else {
                line = readline(prompt);
                if (line)
                        add_history(line);
                return line;
        }
}
```

Problems

- getline is Linux-specific
- Basing only on the interface, it is not clear, who becomes the owner of the return value

Problems

- Presence of the file „QueryParser/Stack.h”
- Why not use std::stack?

```
QueryStats* getQueryStats(string key);
```

Problems

- Who becomes the owner of the result?
- There is no reason for the parameter being passed by value. A const reference would be better.
- This method is just an accessor, it should be const

```
//From Store::NamedItems
if (buffer == NULL)
        debug_printf(*ec, "NULL buffer!!\n");
page_pointer = buffer->getPagePointer(tid, STORE_FILE_, (unsigned int) i);
```

Problems

- Dereference of NULL
- A perfect example of what assigning any logic to NULL values can lead to

```
#ifndef _TRANSACTION_
#define _TRANSACTION_

[...]

using namespace Store;
using namespace LockMgr;
using namespace Logs;
using namespace Errors;
using namespace SemaphoreLib;
using namespace TypeCheck;
```

Problems

- If „using” keyword is used in header files, namespaces stop making any sense – they don't prevent from name conflicts anymore

```
SystemStats::SystemStats(string name) {
        this->name = name;
}
```

Problems

- This simple method has 2 redundant allocations and deallocations!!!
- The argument should be a const reference
- Initializer lists should be used – this way the object is allocated twice

```
namespace TypeCheckConstants {

#define TC_FINAL_ERROR          -112
#define TC_MDN_NAME                     "__MDN__"
#define TC_MDNT_NAME            "__MDNT__"
#define TC_SUB_OBJ_NAME         "subobject"
#define TC_MDN_ATOMIC           "atomic"
[...]
```

Problems

- Defines have nothing to do with namespaces, they are evaluated during the preprocessing phase, so the namespace specification is useless
- These defines pollute the global namespace, it is wrong. Enums inside a namespace should be used. They don't impose any overhead.

Error summary and classification

- Definite errors (least interesting)
- Error proneness
- Readability
- Performance

Definite errors

- Lack of exception safety
- Memory leaks (and other resources too, we used to have semaphores leak)
- Deadlocks (actually a kind of resource)

Error proneness (how to avoid?)

- Use RAII, let the compiler take care of destroying the obejct at proper time – we only specify it in terms of scopes
- Use shared_ptr for common ownership
- Use references to avoid NULL values
- Use dynamic_casts
- Semantics of some operations in C++ is weird at first, watch out for them

Readability

- C++ enables to write very suggestive code, let's make use of it:
  - If you return a reference, hardly anybody will try to free it; if you return an auto_ptr it means that the caller becomes the owner
  - If you use const modifiers you get the full power of the type system in C++

Performance

- Avoid the heap!!!
- Unless you have good reasons for it, don' t pass objects by value
- Use const values, the compiler may optimize the code better
- Use the „virtual” keyword only were it's appropriate, it's not Java, if there is no „virtual” keyword, the compiler has the occasion to inline a function, which may have a **significant** effect
- Use inline consciously, it may both speed and slow the program

Conclusion

- Using C++ features and some widely spread techniques almoast certainly reduces the amount of bugs in the code
- It also increases the readability and performance
- If you don't pay attention to such details, those details will become poor quality of the end product

Thank you

Thank you for listening, any questions or suggestions are welcome, the best place to post them is loxim-devel@lists.sourceforge.net