INTERFACES AND ADAPTERS TO DATABASES USING STANDARD TEMPLATE LIBRARY

Matúš CHOCHLÍK, Karol MATIAŠKO
Department of Informatics, Faculty of Management Science and Informatics, University of Žilina, Univerzitná 8215/1, 010 26 Žilina, Slovak Republic, E-mail: matus.chochlik@fri.utc.sk, karol.matiasko@fri.utc.sk

SUMMARY

This paper presents a set of interfaces and cooperating adapters that greatly simplify the access to datasets, which are the result of a query to a database system in applications developed in the C++ programming language. These interfaces isolate the implementation of the database system from the client code and the adapters allow accessing the records of the dataset, in a more natural way, as native C++ structures possibly using the algorithms from the C++ Standard template library (STL).

Keywords: *Databases, Datasets, Datastructures, STL, Adapters, C++*

1. INTRODUCTION

Since the database systems [5] belong to the most commonly used information systems, it is necessary or at least convenient for software developers to have means to access the stored data in a "friendly" manner, in order to achieve rapid development and easy maintenance of an database application. Well-designed abstraction layer for data access, can save time and other costs when a change in the implementation or even in the type of the database system is needed. Via this layer the programmer can access data which are physically stored in a SQL database, text or binary file located on a local or a network media, etc, without the need to know anything about the details of the actual implementation [7-9].

Because the interface does not change the concrete application code does not need to be rewritten, recompiled and usually not even re-linked, when the implementation of the database system changes. This idea is not new and there were several attempts to develop a standardized interface for data access.

The most popular interface is probably the ODBC: "Open Database Connectivity (ODBC) is a widely accepted application programming interface (API) for database access. It is based on the Call-Level Interface (CLI) specifications from X/Open and ISO/IEC for database APIs and uses Structured Query Language (SQL) as its database access language" [2],[6].

Another important point about developing database applications is, that the data is usually processed in algorithms written by the developer to achieve the desired functionality. The developer therefore must often access the data, i.e. through the ODBC, load them into programming language data structures and only afterwards process them. This step of loading data to language-native structures is very common, thus it would be very helpful, to have another layer, which would access the data, load it into defined structures and present them to the developer in this form.

Because the C++ is one of the most standardized [4] and widespread programming languages, we are going to explore the possibility to develop such layer in it.

In C++ the popular way to store and handle sets of data, is to use the standard template library. According to [1] "The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. "

One of the important things about the STL is that it is defined by the ANSI/ISO standard for the C++ language [4]. This standard defines various high level concepts like general containers, sequences, iterators and algorithms and these concepts are then gradually refined to the level of concrete containers (vector, set, map, list,...), concrete iterators for these containers and concrete algorithms (like find, count, sum, replace, accumulate, sort, and many more). These structures and algorithms are implemented by experts in the field of data structures and algorithms, and with the exception of some marginal cases of use, they are the most efficient implementations of the specific concepts.

On the other hand, the result of a query to a database system is often referred to as dataset. From a programmer's point of view, it contains a set of rows where each row or record can be represented by a C++ structure. So we can think of dataset as of a container of objects, which is very similar to the concepts defined in STL. Furthermore the pointer to a row in a dataset is similar to the concept of an iterator. Most of the STL algorithms operate on containers or iterators and with an appropriate adapter the result of a database query could by passed directly to a basic STL algorithm or to a more advanced, application defined, algorithms.

The goal of this work is to design the interfaces and adapters, which would allow using the results of a database query in a more natural way in the C++ language, compatible with the STL library.

2. SHORT OVERVIEW OF THE BASICS OF STL

The full documentation to STL can be found for example in [1],[4] . The types of objects, that the STL defines, which are of interest to us, are:

2.1. Containers

The definition of the container concept in [1] follows "A Container is an object that stores other objects (its elements), and that has methods for accessing its elements. In particular, every type that is a model of Container has an associated iterator type that can be used to iterate through the Container's elements". From the container are further derived more specialized concepts like Forward Container, Reversible Container, ..., Sequence, Front Insertion Sequence, Back Insertion Sequence, Associative Container, and so on. Then there are defined concrete types of containers like, vector, deque, list, slist, set, map, ..., which are models of these concepts.

2.2. Iterators

According to [1] "Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element." The concept of Trivial Iterator is a basis for Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator and Random Access Iterator. Iterator as a design pattern is also described in [3].

2.3. Algorithms

The STL defines several dozens of basic algorithms of various types including non-mutating and mutating container algorithms, sorting and searching, and various numeric algorithms, which are with the help of the functional objects combinable.

Function objects

[1] Describes function objects as: "A Function Object, or Functor is simply any object that can be called as if it is a function. An ordinary function is a function object, and so is a function pointer; more generally, so is an object of a class that defines operator( )."

The concept of Function object includes Generator, Unary Function, Binary Function, Predicates and various Adaptable functors.

3. CONTAINER AND ITERATOR INTERFACES

The first step to is to create an abstraction layer between the client and the actual implementation of the dataset. This implementation could be anything from the ODBC API, or database libraries, which are specialized to access specific database systems like MySQL or PGSQL, or it could be a simple library accessing data in XML files on a shared network drive. Important for the developer is that the interfaces, which are used to access the data, do not change. These interfaces should have methods similar to the STL concepts; dataset should be derived from the Container and dataset pointer from the Iterator concept. In the section 6, are declared the actual interfaces, which are abstract C++ classes, having similar methods to the methods defined by the STL concepts.

The main difference is that the interfaces will not be parameterized which means that they do not know anything about the type of the data stored in the container or pointed-to by the iterator. The interfaces for container and iterator refer to the data via untyped (void*) pointers, which can lead to several problems if the interfaces are used directly. Special adapters described in sections 4 and 5 solve this issue.

4. TEMPLATE ADAPTERS

One of the most obvious differences between the STL classes and most of the dataset implementations is, that the STL classes are parameterized, which means that they are implemented using templates and that the concrete containers, iterators, algorithms, etc. are specialized to work with a specific data type i.e. a structure or a class. The information about the data type is statically compiled into the code. This has several advantages, like strict type checking which helps to avoid explicit type casting and possible resulting type mish-matches, and also usually leads to better performance.

In the datasets, the data in records are accessed field by field in a dynamic way. The developer can programmatically find out the count and the type or name of the distinct attributes of a dataset. One can also query the values of the fields of a single row of a dataset. This dynamic access has several advantages, but these are rarely needed. In most cases the programmer has created some C++ data structures representing the data in database and then, after the query, she/he iterates through the resulting dataset and loads the data field by field into instances of the defined structure. Much more convenient would be if the result of the query could be adapted to a parameterized container that has information about the C++ data-structure and does the loading from dataset row fields to an instance of the structure automatically.

The resulting design should allow to use both of these approaches. The static, when the user exactly knows the structure of the resulting dataset and wants to use C++ structures containing the data and the dynamic record field browsing when necessary.

5. ADAPTERS TO THE STL

Similar kind of adapters can be used to adapt the abstract interfaces for containers and iterators exactly to the STL concepts. This means that the adapters have all the methods defined by the concepts of Container and Iterator and can be used in STL algorithms.

6. DESIGN

In this section we define the interfaces for abstract containers and iterators, which can be implemented to represent various datasets and dataset pointers.

Interface iContainer, declares basic container functionality:

```
interface iContainer : extends iBase
{
      virtual ~iContainer(){ }
      virtual IIterator GetIterator(void) =
      0;
      virtual bool SetElemCount(int
      paNewCount) = 0;
      virtual int GetElemCount(void) = 0;
      virtual int GetMaxElemCount(void) = 0;
      virtual bool IsEmpty(void) = 0;
      virtual void Clear(void) = 0;
      virtual IIterator Add(void* paPtrVal)
      = 0;
      virtual size_t Remove(void* paPtrVal)
      = 0;
      virtual bool Contains(void* paPtrVal)
      = 0;
};
```

Interface iRevContainer, in addition to a iContainer can return reverse iterator of its elements.

```
interface iRevContainer : extends iContainer
{
      virtual ~iRevContainer(){ }
      virtual IIterator GetRevIterator(void)
      = 0;
};
```

Interface iRAContainer, allows indexed access to its elements.

```
interface iRAContainer : extends iContainer
{
      virtual ~iRAContainer(){ }
      virtual void* GetPtrByIndex(size_t
      paIdx) = 0;
      virtual IBiDiRAIterator
      GetIteratorByIndex(size_t paIdx) = 0;
};
```

Interface iRevRAContainer, is merely a combination of iRevContainer a iRAContainer.

```
interface iRevRAContainer : extends
iRevContainer, extends iRAContainer
{
      virtual ~iRevRAContainer(){ }
};
```

Interface iAsocContainer declares methods for asssociative element storing and retrieval.

```
interface iAsocContainer : extends
iContainer
{
      virtual ~iAsocContainer(){ }
      virtual IIterator Insert(void*
      paPtrVal) = 0;
      virtual void Insert(const IIterator&
      paFirst, const IIterator& paLast) =0;
      virtual size_t Erase(const ISearchKey&
      paKey) = 0;
      virtual size_t Erase(const IIterator&
      paPos) = 0;
      virtual size_t Erase(const IIterator&
      paFirst, const IIterator& paLast)=0;
      virtual size_t Count(const ISearchKey&
      paKey) = 0;
      virtual bool Has(const ISearchKey&
      paKey) = 0;
      virtual IIterator Find(const
      ISearchKey& paKey) = 0;
      virtual IIterator LowerBound(const
      ISearchKey& paKey) = 0;
      virtual IIterator UpperBound(const
      ISearchKey& paKey) = 0;
};
```

There are also interfaces for sequence concepts.

```
interface iSequence : extends iContainer
{
      virtual ~iSequence(){ }
      virtual void* First(void) = 0;
      virtual IIterator Insert(const
      IIterator& paPos, void* paPtrVal) = 0;
      virtual void Insert(const IIterator&
      paPos, size_t paCount, void* paPtrVal)
      = 0;
      virtual void Insert(const IIterator&
      paPos, const IIterator& paFirst, const
      IIterator& paLast) = 0;
      virtual IIterator Erase(const
      IIterator& paPos) = 0;
      virtual IIterator Erase(const
      IIterator& paFirst, const IIterator&
      paLast) = 0;
      virtual void Resize(size_t paCount,
      void* paPtrVal) = 0;
      virtual void Resize(size_t paCount) =
      0;
};
```

```
interface iFISequence : extends iSequence
{
      virtual ~iFISequence(){ }
      virtual void* Front(void) = 0;
      virtual void PushFront(void* paPtrVal)
      = 0;
      virtual void PopFront(void) = 0;
};

interface iBISequence : extends iSequence
{
      virtual ~iBISequence(){ }
      virtual void* Back(void) = 0;
      virtual void PushBack(void* paPtrVal)
      = 0;
      virtual void PopBack(void) = 0;
};
```

Interface iIterator for objects which are used to iterate through the elements of a container.

```
interface iIterator : extends iBase
{
      virtual ~iIterator(){ }
      virtual bool First(void) = 0;
      virtual void Next(void) = 0;
      virtual bool Done(void) = 0;
      virtual ISearchKey GetActualKey(void)
      = 0;
      virtual void* GetActualPtr(void) = 0;
      virtual void SetActualByPtr(void*
      paNewValue) = 0;
};

interface iBiDiIterator : extends iIterator
{
      virtual ~iBiDiIterator(){ }
      virtual bool Last(void) = 0;
      virtual void Prev(void) = 0;
      virtual bool RevDone(void) = 0;
};

interface iRAIterator : extends iIterator
{
      virtual ~iRAIterator(){ }
      virtual void* GetPtrByIndex(int paOfs)
      = 0;
      virtual void SetPtrByIndex(int paOfs,
      void* paNewValue) = 0;
      virtual void Translate(int paOfs) = 0;
      virtual IRAIterator Translated(int
      paOfs) = 0;
      virtual int DistanceTo(const
      IRAIterator& paIterator) = 0;
      virtual bool LessThan(const IIterator&
      paIterator) = 0;
};
```

As we have mentioned, interfaces refer to data using untyped void* pointers. Unfortunately this is necessary, because we don't want to have template interfaces and because of that, we are loosing some of the advantages of C++ type checking. These interfaces should not be used directly, but always through the template adapters, which are wrapping a pointer to an interface and have similar methods like the interfaces but instead of void* they use typed references. In this way we are gaining back some of the type safety. The only issue we must be aware of, is that we must know to what element data type the implementation of iterator or container is referring via the void* and use the correct template adapter.

An example of template wrapper declaration for iIterator. All these wrappers are already implemented and can be easily reused.

```
// this wrapper is actually derived from a
base non-template wrapper where all
// the methods that are data-type
independent are implemented
template <typename tpaDataType> class
IIterator_T : public IIterator
{
public:
      /** constructors destructors of the
      wrapper */
      /** the actual implementation is
      little more complicated because of
      * the error checking, but the idea is,
      that the wrapper method
      * calls the method GetActual of the
      interface and does the typecasting.
      * Using this technique, we then
      implement all the methods of the
      interfaces
      * which are referring to the data
      */
      __INLINE const tpaDataType&
      GetActual(void) const
      {
            return *((tpaDataType*)(this-
               >GetInterface()-
               >GetActualPtr()));
      }
};
```

The real concern to the developer is how to use the wrappers. The wrapper classes implement the -> operator so we can syntactically use them like pointers, when we call methods that are stored-data-type independent, for example the methods First(), Next(), Done(), of the iterator interface. We could also call the other methods and do the typecasting manually, but this is inconvenient so we can call the wrapper methods via the . (dot) operator. An example follows:

```
// assume we have declared a class CData
class CData { ... };
```

```
// this is a getter class which returns to
the client an implementation of iterator

// using the interface iIterator

class CGetter {public: iIterator*
GetCDataIterator(void){ ... } } Getter;

// the actual implementation of the
iIterator returned by GetCDataIterator MUST

// refer to objects of type CData by the
void* in its methods (GetActualPtr(), ...)

// thanks to the constructors of the
IIterator_T wrapper we can do this
initialization

IIterator_T<CData> I = Getter-
>GetCDataIterator();

// we declare also a function which uses an
instance of the CData class

void ProcessCData(const CData& paData);

//the data-type independent methods are
called via the -> operator

I->First();

//the data-type dependent methods of the
wrapper are called via the . operator

while(!I->Done())

{

      //GetActual() Returns a constant
      reference to CData

      ProcessCData(I.GetActual());

      I->Next();

}
```

Because we want to use our abstract interfaces in STL algorithms, we have declared also similar wrappers which are compatible with the STL concepts of containers and iterators.

```
// getter which returns an iterator to
integers according to some type of query

class CValueDB{public: iIterator*
GetValues(iQuery){ ... } } ValueDB;

//

// we declare and initilize three stl
compatible iterators

// IA and IB are iterators to ranges
returned as a result of different queries

// END is a singular iterator (because it is
not initialized) its method

// END.IsSingular() returns true

// If IA->Done() == true (and .IsSingular()
== true) then (IA == END)

// this is necessary to support the STL
ranges, on which most of the algos are
working

IIterator_STL<int> IA(ValueDB-
>GetValues(QueryA));

IIterator_STL<int> IB(ValueDB-
>GetValues(QueryB));

IIterator_STL<int> END;

// some examples of usage in STL algorithms

IA->First(); IB->First();

cout << "Ranges are Equal: " <<
std::equal(IA, END, IB);

//

IA->First();

cout << "Count of ones: " << std::count(IA,
END, 1) << endl;

IA->First();

cout << "Count of tens: " << std::count(IA,
END, 10) << endl;

IA->First();

std::vector<int> V(N); // N > count of
elements of IA

std::copy(IA, END, V.begin());

//

IA->First(); IB->First();

cout << "Dot product: " <<
std::inner_product(IA, END, IB, 0) << endl;

// a little more advanced example using also
some user defined algorithms

IB->First();

cout << "First odd: ";

cout << std::find_if(

        IB, END,

        my_compose2<int>(

              std::equal_to<int>(),

              my_compose2<int>(

                    std::modulus<int>(),

                    std::bind2nd(std::plus
                    <int>(), 1),

                    my_constant<int>(2)

              ),

              my_constant<int>(0)

        )

).GetActual() << endl;
```

We should always have on mind that the iIterator (and iContainer) interfaces can represent very distinct types of iterators (containers), from C/C++ array iterators (actually pointers), file reading classes, XML documents, SQL database data-set pointers, etc. The advantage of this approach is can clearly be seen on the fact that when the implementation of the class behind the interface changes, we don't need to rewrite, recompile or even re-link the code. This advantage is bought at the cost of decreased performance because of the extra de-references and virtual calls, and imposes some potential risks of type-casting related bugs, but in many cases this approach can save a lot of development and maintenance work in cases where the performance of the database system is not very critical.

7. CONCLUSION

The concepts and their implementation presented in this paper should allow much faster designing and development of those database applications, where the flexibility and extensibility is the main goal. The contribution of this work is twofold; First, the abstract interfaces hide the details of the actual

database implementations thus allowing us to do changes of this subsystem without affecting the rest of the source code and second, the cooperation with the STL which allows to use existing algorithms which are fairly common because the STL is widely used in development of applications where processing of datasets is involved.

REFERENCES

[1] Standard Template Library Programmer's Guide http://www.sgi.com/tech/stl/
[2] Microsoft Developer Network http://msdn.microsoft.com/library/
[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns – Elements of Reusable Object-Oriented Software, Pearson Education, Inc. Addison-Wesley Professional 1995
[4] Dirk Louis, Petr Mejzlík, Miroslav Virius, Jazyky C a C++ podle normy ANSI/ISO, Grada Publishing, 1999
[5] Matiaško, K.: Databázové systémy, EDIS 2002
[6] WIKIPEDIA, http://en.wikipedia.org/wiki/Odbc
[7] Kollár, Ján: Paralelné programovanie. Academic Press elfa, s.r.o., Košice, 1999, monografia, ISBN 80-88964-14-8, 96pp.
[8] Kollár, Ján: Metódy a prostriedky pre výkonné paralelné výpočty. Academic Press elfa, s.r.o., Košice, 2003, Edícia monografií FEI TU v Košiciach, 110 pp., ISBN 80-89066-70-4
[9] Kollár, Ján: Process Functional Programming. Proc. 33rd Spring International Conference MOSIS'99 - ISM'99 Information Systems Modelling, Rožnov pod Radhoštěm, Czech Republic, April 27-29, 1999, ACTA MOSIS No. 74, pp. 41-48, ISBN 80-85988-31-3

BIOGRAPHIES

Matúš Chochlík was born in 1981. In 2005 he graduated in the study field of Information and Management Systems at the Faculty of Management and Informatics of the University of Zilina. Since 2005, he is working like a PhD student at this faculty. His research and employment activities have been oriented in the field of object-oriented and distributed databases, simulation and computer graphics.

Karol Matiaško was born in 1952. In 1975 he graduated in the study field of Cybernetics at the Faculty of Electrical and Mechanical engineering of the University of Transport and Communications in Žilina. He received PhD in Technical Cybernetic in 1988 at University of Transport and Communication in Žilina like employee of the Research Institute of Transport. Since 1990 he was joined with the Faculty of Management Science and Informatics of University of Žilina and his research and educational activities have been oriented in the area of Database system, Distributed processing and Data processing.