Molecular Data Structure in Generic Algorithm

Wei Zhang[a], Tingjun Hou[b], David A. Case[*a], Xiaojie Xu[c]

([a]Department of Molecular Biology, The Scripps Research Institute, La Jolla CA 92037, USA)

([b]Department of Chemistry and Biochemistry, University of California at San Diego, La Jolla, CA 92093, USA)

([c]College of Chemistry and Molecular Engineering, Peking University, Beijing 100871)

Abstract: Here we proposed a template library, named Molecular Objects and Relevant Templates (MORT), suitable for both computational programs and user interfaces. Three basic concepts in MORT are "parameter", "component" and "adjacency". A parameter is a property of a chemical object, for example, atomic number, position and velocity components are all parameters, for each parameter, an individual data type is defined, and template functions are used to access parameters, so that new parameters can be easily introduced at any time. A component is the container of a set of parameters, data of same parameter type is stored together for best efficiency, atom, bond, residue are most typical components. An adjacency records the relationship between two component . Hence, in MORT, molecule is a set of components and the adjacencies between them, and atom, bond, residue are iterators pointing to data inside a component. Two model programs were designed to test MORT's efficiency, extensibility and flexibility. One is a new graphical user interface for AMBER named gleap, and the other is a Possion-Boltzmann equation (PBE) solver program. The results are satisfactory and encouraging.

* Correspond author, e-mail: case@scripps.edu, phone: +1-858-558-9768

INTRODUCTION

There are two kinds of chemical programs, computational programs and user interface (UI) programs. They are different in many ways, for computational programs, data structure is usually simple, and developers tend to use FORTRAN to get best efficiency. For example, in PBE solvers and ab initio programs, a molecule is just a set of atoms, the bonds between atoms are not important, which explains why the widely used PBE solver Delphi and ab initio programs Gaussian are coded in FORTRAN77. On the other hand, in a UI program like antechamber[2], data structures are more complicated, a molecule must include atoms, bonds, residues, subsets and their relationships so that it can do atom type assignment. Thus, UI programs are usually developed using the objected-oriented languages like C++ for its power on data structure representation.

Problem is computational programs are growing more and more complicated in the last decade, and FORTRAN77 cannot handle this complexity easily, which stimulate scientists to find better languages in computational program development. For example, AMBER[3] developers chose FORTRAN90 and developers of NAMD[4] chose C++. Meanwhile, with the development of compilation technique and the birth of some new technique (such as expression template[5]), C++ gains a great improvement in its efficiency and is considered to be as fast as FORTRAN now.[6] In Veldhuizen's paper,[6] they compared the speed of C++, FORTRAN77 and FORTRAN90, and found in many aspects C++ is as fast as FORTRAN, and even faster then FORTRAN90.

Thus now is time for us to design a data structures representation of chemical objects for general usage. Some effort has been tried by us before about using template to build up the whole data structure[7]. But we finally found out that this idea was not applicable under the current circumstance that the separate compilation of template has not been supported, since it would cause unbelievably long time to compile a program, and a minor revision on a generic algorithm would cause a huge re-compilation. Besides, it is hard to debug template functions. However, the idea of

declare a new data type for each parameter and access parameter via template functions is still right. But in the library reported here, the usage of template has been restricted to a very low ratio, and the name of our library has also been changed from Molecular Handling Template Library (MHTL) to Molecular Objects and Relevant Templates (MORT). In the following sections, we will discuss the realization of some important objects in the library.

DATA STRUCTURE

As has mentioned before, three most important concepts in MORT are parameter, component and adjacency:

Parameter. A parameter is a property of a chemical object. For example, atomic number, position, partial charge is atom's parameter, and bond order is bond's parameter. In MORT, an individual data type is declared for each parameter, and there is a template class parm_T can be used to ease the procedure, the declaration of parm_T is like:

```
template< typename T, char c0, char c1='\0, char c2='\0', ....., char c50='\0'> class parm_T,
```

the first template argument T is actual type of parameter, and other arguments are parameter name, and parameter types can be declared as:

```
typedef parm_T< double, 'c', 'h', 'a', 'r', 'g', 'e' > charge_t;

typedef parm_T< int, 'o', 'r', 'd','e','r'> order_t;
```

Component. Component is container of a set of parameters. Almost all the chemical objects can be represented as a component, such as atom, bond and residue, they just have different parameters. The declaration of class component_t is like:

```
class component_t

{

public:

        component_t();

        component_t( int size);

        virtual ~component_t();

        template< typename T > vector< T::valut_t >::iterator begin();

        template< typename T > vector< T::value_T >::iterator end();

        template< typename T > T::value_t& at( int id );

        template< typename T > vector< T::valut_t >::const_iterator begin() const;

        template< typename T > vector< T::value_T >::const_iterator end() const;

        template< typename T > T::value_t const& at( int id ) const;

        int add();

        void remove( int id );

private:
```

```
        hash_map< int, holder_i* > m_parms;

        vector< int > m_ids;

};
```

The methods to access parameter have been discussed elsewhere[7]. Here the dynamic access method has been taken. A base class is declared as the following:

```
class holder_I
{
    virtual shared_ptr< holder_i > clone() = 0;

    virtual void resize( int size ) = 0;

};
```

Then a template class named holder_T inherited from holder_i stores the variable in it. The difference from original method is that holder_T holds a variable's array in it instead of a single variable in order to obtain better memory impact and higher efficiency. There is a member variable of the type hash_map< int, holder_i*> in class component, and it uses property's identical number as hash code to stored pointer of holder_T in it. The pointer can be casted back into its original type using RTTI (Run Time Type Identification) mechanism of C++.

As can be seen from the above, component is a set of objects, each object is identified by an ID number, which is stored in an ID array and used to access parameter. When member function add is called, an new ID will be generated, stored in m_ids and then returned to caller, call member function remove with an ID number will remove the ID from m_ids, and m_parms remain the same. It is because data are stored as linear table in m_parms for its efficiency, and the

shortcoming of a linear table is it is hard to delete an element from them, you will need move all the rest elements, which will be very time-consuming.

Adjacency. An adjacency records the relationship between two components. The declaration of adjacency_t is like the following:

```
class adjacency_t

{

public:

        adjacency_t();

        ~adjacency_t();

        bool add( int a, int b );

        bool has( int a, int b ) const;

        bool remove( int a, int b );

        iterator find ( int a, int b ) const;

        iterator begin( int id ) const;

        iterator end  ( int id ) const;

        int size( int id ) const;
```

```
private:

    vector< vector< int > > m_adjs;

}
```

As has mentioned above, objects in a component are distinguished by an ID number, thus call add with two objects' ID number will register a relation between them, and this relation can be test, find, and remove later via member function has,find and remove. Member function begin and end can be used to enumerate relations of a given object, size will give the number of relations of an object.

Other important classes in MORT include molecule_t and object_T, iterator_T:

Molecule. Molecule contains a set of components and the adjacencies between them, and provides an interface them. The declaration of molecule_t is like the following:

```
class molecule_t

{

public:

    molecule_t();

    virtual ~molecule_t();
```

```
        bool add_component( int compid, int size = 0);

        bool has_component( int compid );

        component_t* get_component( int compid );

        bool remove_component( int compid );

        bool add_adjacency( int comp_a, int comp_b );

        bool has_adjacency( int comp_a, int comp_b );

        adjacency_t* get_adjacency( int comp_a, int comp_b );

        bool remove_adjacency( int comp_a, int comp_b );

    private:

        hash_map< int, component_t* > m_components;

        hash_map< int, adjacency_t* > m_adjacencys;

    };
```

each component has a ID number for identification, so does each adjacency.

Object. Object_I is a template class which provide an easy method to access object's parameters. Currently to access an atom or a bond programmers need to first find their correspond component, then find the parameter by objects ID number sequence number. Template class object_I is designed to ease the procedure, the declaration of object_I is like:

```
template< int comp > class object_I
{
public:
    object_I( molecule_t& mol, int oid );
    template< typename T > void set( const T& parm );
    template< typename T > T::value_t get( );
    virtual int getoid() const;
private:
    molecule_t* m_pmol;
    int m_oid;
};
```

template argument comp is ID of component, and inside the class there is pointer to molecule, and object id, with the three, object_I can access the parameter of a designated object easily. Several commonly used object type is declared as the following:

```
typedef object_I< ATOM > atom_i;

typedef object_I< BOND > bond_i;

typedef object_I< RESD > resd_i;
```

Moreover, object type has another usage, it can be used to handle related parameter via template specialization. Related parameter is some parameter related to other. For example atomic symbol is related to atomic number, thus when user try to set atomic symbol, the ideal behavior would be first get the correspond atomic number from a table then set atomic number parameter, and when user try to get atomic symbol, it will find the symbol according to the atomic number from a table. This ideal behavior can be achieved via the technique of template specialization, the following is the example code:

```
template<> void atom_I::set< symbol_t >( const string& sym)
{
    set< element_t >( to_element( sym ) );
}
template<> string atom_I::get< symbol_t >( const string& sym)
```

```
{
    return to_symbol( get< element_t >() );
}
```

Iterator. Template class iterator_T provide the ability to traverse on a data structure. One problem of object_I is it unlike a pointer, it can not be used to iteratively, i.e., to set the atomic number of all atom to 1, we need code like the following:

```
for( int i=0; i < atom_number(mol); ++i )
{
    atom_i atom( mol, 0);
    atom.set< element_t >( 1 );
}
```

which is tedious. But object_I do left a method for this, it is the virtual member function getoid, which is used to get object ID number, in object_I it just simply return m_oid, but other class inherited from object_I can re-implement this function, to provide iterative function. It is what iterator_T does. The following is the declaration of template class iterator_T:

```
template< int compid, typename iter_t >
class object_Iterator : public object_I< compid >
```

```
{
public:
        object_i& operator*();

        object_i const& operator*() const;

        object_i* operator->();

        object_i const* operator->() const;

        virtual int getoid() const;

        virtual int& mut_index();

        iterator_t& operator++();

        iterator_t& operator--();

        iterator_t operator++( int );

        iterator_t operator--( int );

        iterator_t& operator+=( int diff );

        iterator_t& operator-=( int diff );

        friend int operator-( const iterator& lhs, const iterator_t& rhs );

        friend bool operator<( const iterator& lhs, const iterator_t& rhs );
```

```
        friend bool operator>( const iterator& lhs, const iterator_t& rhs );

        friend bool operator<=( const iterator& lhs, const iterator_t& rhs );

        friend bool operator>=( const iterator& lhs, const iterator_t& rhs );

       friend iterator operator+(const iterator_t& lhs, int diff );
       friend iterator operator-(const iterator_t& lhs, int diff );
};
```

as can be seen from the above, iterator_T is implemented as a pointer to a object_T, it has the ability to traverse on a container, as long as the contain has its iterator. Some most commonly used iterator is declared as the following:

```
typedef iterator_T< ATOM, vector< int >::iterator > atom_t;

typedef iterator_T< BOND, vector< int >::iterator > bond_t;

typedef iterator_T< RESD, vector< int >::iterator > resd_t;
```

and some related functions have been provided:

```
atom_t atom_begin( const molecule_t& mol );
atom_t atom_end( const molecule_t& mol );
bond_t bond_begin( const molecule_t& mol );
bond_t bond_end( const molecule_t& mol );
resd_t resd_begin( const molecule_t& mol );
resd_t resd_end( const molecule_t& mol );
```

with these types and functions, the code in the beginning of this section can be rewritten as:

```
for_each( atom_begin( mol ), atom_end(mol), bind( &atom_i::set< element_t >, _1, 1 ) );
```

Besides the usage of standard types atom_t, bond_t and resd_t, programmers are encouraged to develop their own iterator types, for example. If a set of atom ID numbers are stored in a set< int > structure named idset, and programmer want to set these atoms' atomic number to 1, the following code can be used :

```
typedef iteator_T< ATOM, set<int >::iterator > atom_set;

for_each( atom_set( mol, idset.begin()), atom_set( mol, idset.end() ), bind( &atom_i::set< element_t >, _1, 1 ) );
```

Algorithm

The algorithm part of MORT is almost same to that of MHTL. Three types of subroutines are provided, including: (1) molecular file IO routines, (2) SMARTS language based atom typing routines and (3) molecular OpenGL rendering routines.

Model Programs

After this whole system has been set up, two model programs have been developed to testify its efficiency, extensibility and flexibility. The first program is a PBE solver, a typical computational program, and the second program is a graphical user interface for AMBER named gtkleap.

PBE Solver. The PBE solver was developed according to Delphi's algorithm. Two components were employed in the molecule. One is the ATOM component, has position, partial charge and VDW radii as its parameters. The other is the GRID component, has its partial charge and dielectric constant as its parameters. The program gives same reaction field energy as Delphi. The efficiency of this program has been tested on two standard systems: BPTI and SOD, the result is shown in Table 1. As it can be seen from the table, our program is as fast as Delphi.

Gleap. Gleap was designed to be a replacement of current amber UI xleap. It has a command window and a display window. User can input command in the command window, and the molecule in the display window will be adjusted according to user's input.

Gleap uses Model-View-Control (MVC) pattern which is widely used by UI programs. The three main classes of gleap are content_t (the model), display_t(the view) and control_t(the control). The declaration is like the following:

```
class content_t

{

public:

    content_t();

    virtual content_t();

    bool add( const string& name, molecule_t* mol );

    void remove( const string& name );
```

```
        iterator begin();

        iterator end();

    private:

        hash_map< string, molecule_t* > m_data;

};
```

content_t (also named database_t) is basically a wrapper of hash_map< string, molecule_t* >, provides add, remove, iterate ability. All the molecules stores here.

Class display_t

```
{

public:

        display_t();

        virtual ~display_t();

        void add( const string& name, graphic_i* graphic );

        void remove( const string& name );
```

```
    void resize();

    void repaint();

private:

    hash_map< string, graphic_i* > m_graphics;

};
```

display_t holds all the functions related to view.

```
Class control_t

{

public:

    control_t();

    virtual ~control_t();

    static void add( const string& name, command_i* command );

    void run( const string& command );

private:
```

```
        static hash_map< string, command_i* > m_commands;

};
```

control_t is the main processor of Gleap. The member funciton add is used to register a command to command array. The member function run parse a command line into arguments and call correspond command with given arguments.

Another important class is console_t, which is used to handle keyboard events. The declaration is like:

```
class console_t

{

public:

     void up_pressed();

    void tab_pressed();

    void down_pressed();

    void enter_pressed();

}
```

As it can be implied, the molecule used in Gleap is much more complicated that the one used in PBE solver. It has the

component of atom, bond and residue, and the adjacencies between them. Moreover, it is growing more complicated when adding new command. The picture illustrated in Figure 1 is the output of "ribbon" command of gleap, which uses OpenGL NURBS subroutine to display secondary structure, a new component ribbon to added to store all the control points.

Conclusion

Here we present a design of the most used data structures in the chemical software, the structure of molecule, atom, bond, residue and their relationships. A lot of new techniques have been used such as generic programming, hash map, template meta-programming, object oriented design to make sure of its efficiency, extensibility, flexibility and transferability, and these characters have testified by two model programs: a PBE solver which is a typical computational program and a GUI which is a typical user interface program.

Acknowledgement

This research is supported by National Scientific Fund of China, No. 20375002. The source code of MORT can be obtained upon request.

REFERENCES

1. Gilson M.K., Sharp K. and Honig B. Calculating the Electrostatic Potential of Molecules in Solution: Method and error assessment. *J. Comp. Chem*. **1987**, *9*, 327-335.
2. InsightII 2000, Accelrys. http://www.accelrys.com.
3. Case D.A., Darden T.A., Cheatham T.E. III, Simmerling C.L., Wang J., Duke R.E., Luo R., Merz K.M., Wang B., Pearlman D.A., Crowley M., Brozell S., Tsui V., Gohlke H., Mongan J., Hornak V., Cui G., Beroza P., Schafmeister C., Caldwell J.W., Ross W.S., and Kollman P.A. AMBER 8, University of California, San Francisco, **2004**.
4. Bhandarkar M., Brunner R., Chipot C., Dalke A., Dixit S., Grayson P., Gullingsrud J., Gursoy A., Hardy D., Humphrey W., Hurwitz D., Krawetz N., Nelson M., Phillips J., Shinozaki A., Zheng G., Zhu F. NAMD2.5, Theoretical Biophysics Group, Beckman Institute, University of Illinois.
5. Veldhuizen T.L. Expression templates. C++ Report, **1995**, *7*, 26-31.
6. Veldhuizen T.L., Jernigan M.E. Will C++ be faster than Fortran? Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'97).
7. Zhang W, Hou TJ, Qiao XB, Xu XJ. Some basic data structures and algorithms for chemical generic programming. *J. Chem. Inf. Compt.* **2004**, *44*, 1571-1575.

Table 1. The comparison between our PB solver and Delphi

System	Scale(grid/Å)	Time(s)	
		Our PB Solver	Delphi
SOD	0.8	0.29	0.21
	2.0	3.93	3.76
PTI	0.8	2.37	2.07
	2.0	48.54	46.95

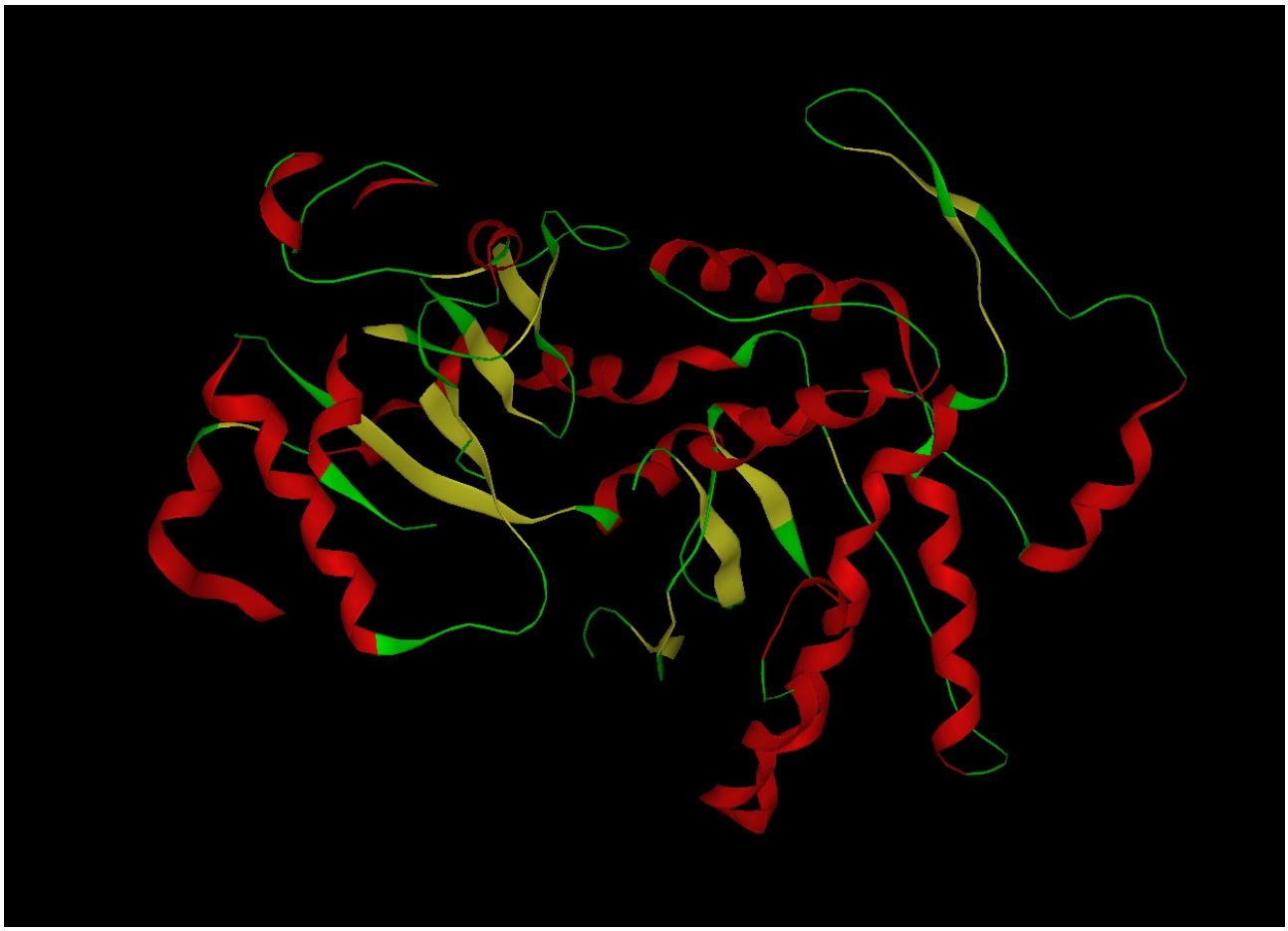

Figure 1. Screen shot of Gtkleap