gLEaP Programmer's Guide

Wei Zhang
The University of Texas Health Science Center at Houston
6431 Fannin St
Houston, TX 77030

February 25, 2009

Chapter 1

Introduction

Welcome to leap's world! This document is provided for people who are interested in gleap and want to eventually contribute to it.

In this chapter, we introduce background knowledge of gleap. Firstly, we will explain how the source code of gleap is orgnized; then comes a short introduction of C++ which is the programming langugage we use to develop gleap; then we will spend some time introduce the boost library, which is a multi-functional C++ library widely used in our development.

1.1 Code Orgnization

There are currently the following sub-directories under gleap source tree:

mortsrc/	source codes of mort, an C++ library used for all chemical operations.
mortest/	test cases for mort
leapsrc/	source codes of gleap
leaptst/	test cases of gleap
plugins/	source codes of all the plugins
freelib/	contains all the external GPL libraries used by gleap, boost and gtkglext
example/	examples shows how to code mort and gleap

As you might have noticed, all these directorie have 7-charactor names, which is not a coincidence but a result of choice, since the authors believes it looks better, and will be very appreciated if future developers keep this rule.

The first five directories are of most importance and we will come back later to introduce it. Right now let's take some time to explain some C++ language features gleap code has used.

1.2 C++ Q&As

It is our impression that most readers of this document are long time C and fortran programmers but are not very familiar with C++. Unfortunately, gleap is written in C++ and uses a lot of its features which is not included in C and Fortran, such as template and RTTI (Run-Time Type Identification). There is no intention to introduce C++ in such a short section, here we will just list some questions about C++ which we think readers might will come up with when they are reading gleap source code, and their answers. Here we assume readers already knew some basic concepts of C++, such as class, constructor, copy constructor, assignment operator. If you don't, please refer to your favourite C++ text book for a full description.

What does a `&` mean in a statement like `int& idx`?

It is declaration of a reference. Reference is a new feature in C++, which in concept is fairly close to hard link of unix filesystem. As you know, whatever operations has performed on a hard link is performed on the file it pointed to, so does a reference. Though it is usually implemented via pointer, reference is not identical to pointer. A major difference is pointer could be null (means it can be not pointing to anything) but reference can never be empty, i.e. it must be pointing to something. An statement like: `int& ridx;` won't compile. These characteritics makes it perfect to be function's argument.

For example the following C++ code:

```
void swap( int& a, int& b )
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a=3, b=1;
    swap( a, b );
}
```

is identical to C code:

```
void swap( int* pa, int* pb )
{
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

int main()
{
    int a=3,b=1;
    swap( &a, &b );
}
```

just C++ code saved your energy to get address via `&` and refer via `*`. I am pretty sure you will love this feature, after you write any function longer then 100 lines.

What does `template<...>` suppose to mean?

It means declaration of template class or template function. (Template is a fairly new feature introduced to C++, thus there might be a chance that your C++ text book doesn't cover it, in this case throw it and get a new one.) It is a type argumented technique which allows you to use a type as argument when defining classes and functions.

Template in my opinion is the greatest thing that ever happened to C++. For example, with template we can define a unified `abs()` function for all the arithmetic types, like the follwing:

```
template< typename T >
T abs( const T& v )
{
    return (v>0) ? v : -v;
}
```

and absolute value of any type variable can be obtained as long as they have a '<' operator and a '-' operator defined. In C, this kind of job is usually be done via Macro, which is dangerous, type unsafe and undebugable.

The function of template are much more than that, based on template, a set of frequently used containers has been written, such as linear table, deque, stack, double linked list, set, map, multiset, multimap, and algorithms has also be written, such as find, copy, remove, etc. These containers and algorithms forms STL (the Standard Template Libary), which is now part of of C++ standard. A new programming paradigm has been established, which is called "Generic Programming".

Moreover, with template specialization and partial specialization technique, a new form of programming language "Meta Programming Language" is possible. Please refer to the book *Modern C++ Design* and documentation of *boost::mpl* for more details of MPL.

What does a `vector<...>` mean?

You already know `<..>` is template. `vector` is the STL implementation of linear table. It is the one you should use to replace traditional array in C++. The C array has a lot of limitations, firstly, its size is fixed once declared, secondly, it is allocated on stack so the size is limited. `vector` does not has these limitations. For example:

```
        vector<int> a(10);
```

is identical to

```
              int a[10];
```

We all have experienced the problem of predefining the size of array in C programming, too small will cause buffer overflow, too big will cause memory waste. Some choose to use dynamic allocation:

```
           int* a = malloc( num_atom * sizeof(int) );

           ...............

           free( a );
```

despite the complicity of the code and the exceptional safety problems it has (if something bad happened between malloc and free cause program exit abnormally, the memory region assigned to a is leeked), it still can not handle the case of "growing array problem" means we want to increase the size after it is allocated. Now with vector, all these problems mentioned are gone, and all you need to do is declare the array as:

```
           vector<int> a( num_atom );
```

Then you can do anything you want on array a. If program exit abnormally, C++ standard assure you that all objects allocated on stack, their destruction function will be called, which means no memory leak will happen. and you can allways add a new element to the back of a by using its member function `push_back`. If you want pass the pointer of an array to some C function, you can do that by getting the address of first element of `vector`, C++ standard assure you everything in vector is stored linearly.

1.2.1 What about *const*?

`const` is a qualifier which when used on variable could protect the variable from being modified. Once a variable is declared as `const`, its value can no longer be changed. For example, the following code won't compare:

```

    const int a = 3;
    a = 4;
```

`const` is in most cases used in function argument declaration. All arguments if their value is not supposed to be changed inside the function should be declared as `const`. `const` could help programmers from making some stupid mistake. Here we show an example:

```

    void do_something{const int& tag)
    {
        if( tag = A)
        {
            do_A();
        }
        else if( tag == B)
        {
            do_B();
        }
        else
        {
            do_C();
        }
```

The above code has a typo at line 5, it should be `if(tag==A)`, instead of `if(tag=A)`, it will be discovered at compile time since `tag` has been declared as `const`. The compiler will popup an error message complaining something about "discard qualifier". It won't be that easy if `tag` has not been declared as `const`, it may take hours before you find this. This is the real value of const.

`const` can be complicated when used on a pointer or an object. When used on a pointer, it could mean the pointer itself can not be changed or the pointee (the thing poiner is pointing) can not be changed depending on it appear after or before the symbol `*`, as show in following example:

```
    char str1[] = ``Hello, world!''
    char str2[] = ``Good moring!''


    const char* ptr1 = str1; // pointee is const, pointer is mutable
    ptr1[0] = 't'; // wrong, won't compile since pointee is const
    ptr1 = str2;  // OK, since pointer itself is mutable

    char const* ptr2 = str1; // pointee is const
    ptr2[0] = 't'; // wrong, won't compile since pointee is const
    ptr2 = str2;  // OK, since pointer itself is mutable

    char * const ptr3 = str1; // pointee is mutable, pointer is const
    ptr3[0] = 't'; // OK, since pointee is mutable
    ptr3 = str2; // wrong, won't compile since pointer is const
```

`const` when used on an object or a reference to an object means only const member function can be called. Yes, a member function can be declared as `const`, means it can be used for const object, in such a member function, all the data members the object is treated as `const`. Look the following code for instance:

```
    class foo
    {
    public:

        void set(int a) // can't be const, since it changes data member's value
        {
            m_data = a;
        }

        int& get() // can't be const, since it returns reference of internal
                   // data member which can later be used to change the data
                   // member
        {
            std::cout << ``mutable get is called!'' << std::endl;
            return m_data;
        }

        const int& get() const // be const since it returns a const reference
        {
            std::cout << ``const get is called'' << std::endl;
            returm m_data;
        }

  private:

        int m_data;
    }

    void func1(foo& f)
    {
       f.set(1);      // OK, since f is mutable
       f.get() = 2;   // OK, since f is mutable, should print ``mutable get is called''
       int b = f.get(); // OK, should print ``mutable get is called''
    }

    void func2(const foo& f)
    {
        int b = f.get(); // OK, should print ``const get is called''
    }
```

1.2.2 What is a `XXX::iterator`?

`XXX::iterator` refer to a nested type iterator defined inside the container class XXX, which is used to traverse on XXX instances. The typical usage of an `XXX::iterator` is demonstrated in the following code:

```
        XXX a;

        XXX::iterator i = a.begin();
        for( ; i != a.end(); ++i )
        {
            foo( *i );
        }
```

Iterator is close to pointer in usage, but it is more general since a pointer can only operate on linear memory regions, while iterator can be implemented to traverse on any kind of container such as linked list and map.

Iterator is extremely important for generic programming. To understand that, it need to be explained how GP works, basically, we setup a concept first which include some requirements, and write generic algorithms (usually in the form of template function) based on this concept, meanwhile data structures fit for these requirements are developed, which are called models of the concept. Then the generic algorithms can operate on these models. An example might help to understand how it is working. see we want to find a certain value in a `vector<int>`, a `int[10]`, or a `string`, what we will do is write down the following algorithm find:

```
    template<typename iter,typename T>
    iter find(iter first,iter last,const T& t)
    {
        while( first != last&&!(*first==t) )
            ++first;
        return first;
    }
```

and use it like the following example:

```
    vector<int> a(10);
    vector<int>::iterator ia=find(a.begin(),a.end(),1);
```

```
double b[10];
double* ib=find(b,b+sizeof(b)/sizeof(double),1.0);
```

BTW, the code mentioned in the beginning of the subsection can be rewritten as:

```
XXX a;
for_each(a.begin(),a.end(),ptr_fun(foo));
```

1.2.3 What is a `shared_ptr<...>`?

`shared_ptr<...>` is used to replace pointer as the handler of an allocated memory area. In C, the following code is used to allocate new memory:

```
        int* a = malloc( 10 * sizeof(int) );
              .......
        free( a );
```

and the correspondant C++ would be:

```
        vector<int>* a = new vector<int>( 10 );

              .......

        delete( a )
```

both use a simple pointer to handle allocated memory reigon, which is not exceptional safe, as we already mentioned before (image what will happen if something goes wrong and causes an abnormal exit). and there is some more problems with this memory management strategy. All the design rules tell us that new and delete should appear pairly. What if some function must return a new object? Then who should be responsible for the deletion. All these problems would not even exist if you choose to use `shared_ptr<...>` to handle your allocated memory region. What `shared_ptr<...>` will do is hold the pointer inside the class and maintaining a reference count on this pointer. The counter increases when a new `shared_ptr<...>` is constructed and start sharing the pointer, and decreases when a `shared_ptr<...>` sharing the pointer is destructed. After all, when the counter goes to zero, which means no more `shared_ptr<...>` is pointing to this area, delete will be called and the memory region is release. All these process are automatic, all you need to do is assign the allocated region to a `shared_ptr`. There is no need for you to call delete by yourself. Actually, if you are familiar with java, you will noticed `shared_ptr<...>` is almost same to java's pointer.

The following code demonstrates the usage of `shared_ptr<...>` and explains how it works:

```
shared_ptr<string> p1=new string(``Hello, world!'');
//now p1=> ``Hello,world'', ref_count=1

shared_ptr<string> p2=p1
//now p1,p2=> ``Hello, world!'' ref_count=2

p1.clear();
//now p1=NULL, p2=>''Hello, world!'', ref_count=1

p2.clear();
//now p1=NULL, p2=NULL, memory is released.
```

What is `std::for_each`

In gleap source code, sometimes you will see statement like the following:

```
std::for_each(begin,mol.end, func);
```

and you are wondering, what is this `std::for_each`, `for_each` is one of these stl algorithms which *for each* item in the given range (`[begin,end)`), call `f` and use the item as argument. Thus, it can be translated to the following code:

```
  for( iter i=begin; i != end; ++i )
  {
      func( *i );
  }
```

There are a lot of STL algorithms, such as `copy`,`remove`,`remove_if`,`sort`, check any C++ text book for the full list of them. STL algorithms are really important features of C++, and it will be more powerful when combined with functors.

An interesting fact of STL algorithms is there is no such algorithm `copy_if`, (there is something called `remove_copy_if`). The truth is people in C++ standard commitee forget to put it into C++ standard in 1998, it is so important that we put it into gleap source code. Hopefully they will remember to put in the new C++ standard (maybe C++ 07'?).

What is a functor?

We have mentioned the term "functor" in our previous section, and you might have heard about it before. So what it a functor? basically functor is a class (or struct) which has implemented `operator()` thus works somehow like a function.

In the following code we illustrates a functor which test if an integer number is greater than 3, and how to use the functor.

```
struct greater_than_3_t
{
  bool operator()( int input )
  {
    return input > 3;
  }
}

greater_than_3_t greater_than_3;

assert( greater_than_3(5) );
assert(!greater_than_3(1) );
```

A tricky thing of functor is, it is a class, thus you must first declare an instance which then can be used just like a fucntion.

You might will ask why we even bother using a functor, in our example, why don't we just declare the following function `greater_than_3_f`:

```
bool greater_than_3_f(int input)
{
  return input > 3;
}
```

which will work as same as `greater_than_3_t`.

Firstly, functor is in most cases used with STL algoirithms. For example, the following code find the first element in a `vector<int>` that is greater then 3:

```
  vector<int> iv;
    ....

  vector<int>::iterator i = find_if( iv.begin(), iv.end(), greater_than_3_t() );
```

please pay attention to the parentheses after `greater_than_3_t`, it means declaring an instance. Even with this example, we can not prove why functor is better than function, since using function we can archieve the same effect:

```
    vector<int> iv;
      ....
    vector<int>::iterator i = find_if( iv.begin(), iv.end(), greater_than_3_f );
```

but what if we want find element that is greater than 5? via function, we must write another function `greater_than_5_f`, but if we use functor, we can modify our functor, let it take an argument in constructor then give an general solution, here comes our new functor `greater_than_t`:

```
  struct greater_than_t
  {
    typedef argument_type int;
    typedef return_type bool;

    greater_than_t(int rvalue)
    {
      m_rvalue = rvalue;
    }

    bool operator()(int input)
    {
      return input > m_rvalue;
    }

    int m_rvalue;
  };
```

and in the following example codes, first line find the first element greater than 3, the second line find the first element greater than 5 in the rest elements:

```
  vector<int> iv;
  ...

  vector<int>::iterator i = find_if(iv.begin(), iv.end(), greater_than_t(3) );
  vector<int>::iterator j = find_if(j+1,        iv.end(), greater_than_t(5) );
```

but if we use function, we have to use two functions, which usually cause code duplication.

Moreover, functor can be combined with functor adaptors. For example, if we want find an element which is less than or equal to 3, we do not need to write new functor, we just use the functor adaptor `std::not1`, as the following example:

```
   vector<int> iv;
     ....

   vector<int>::iterator i = find_if(iv.begin(), iv.end(), std::not1( greater_than_t(3) ) );
```

The functor adaptor `not1`, as the name suggested, invert the result of a functor. Here the number 1 indicates number of arguments. Refer to your favorite C++ text book for a full list of STL functor adapotrs, as an hint, adaptor `std::ptrfun` convert a function ptr to a functor.

These are all we can come up with right now, I would be more than happy to answer questions in this area and put that into the list.

1.3 Introduction to Boost

You may have noticed that under the directory freelib, there is directory named boost, which contains a distribution of boost library version 1.33.

Boost is a C++ library contains a lot of useful algorithms and classes. Here is a list which has been used in gleap.

1.3.1 boost::any

One thing that C++ different from other pure OO language is C++ does not have a single-rooted inheritence system. It has its historitic reason, and has some disadvantages. One of them is it is not as easy as in Java to store everything in a media and turn it back later. For example, the following Java code has no obvious corespondace C++ code:

```
      int ia = 3;
      Object media_i = ia;
      int ib = (int)media_i;

      String sa = new String( ``Hello, world'' );
      Object media_s = sa;
      String sb = (string)media_s
```

BTW, the code above is not as simple as it looks, since int is primitive type in Java and is not inherited from Object, what really happens is int turn into Integer which is inherited from Object. This procedure is so-called “boxing”.

To obtaining the same effect, such a single root inheritence system need to be established, and it can be achieved using template and RTTI techniques. Boost::any is an implementation of the procedure. The following code demostrates its usage:

```

int ia = 3;
boost::any any_i = ia;
int ib = boost::any_cast< int >( any_i );
assert( b == 3 );

string sa = ``Hello, world'';
boost::any any_s = sa;
string sb = boost::any_cast< string >( any_s );
assert( sb == ``Hello, world'';
```

When any is casted to another type other than the one it is assigned, a `bad_cast` exception will be throw. It is interesting and worthwhile to explain how `boost::any` works. Firstly an base class `holder_i` is defined, which is later used by any as the root class:

```

class holder_i
{
public:

    virtual shared_ptr< holder_i > clone() const = 0;
}
```

Then a template class `holder_T` is implemented inheriting from `holder_i`, which have the real data stored:

```

    template< typename T >
    class holder_T : public holder_i
    {
    public:
        holder_T( const T& rhs )
            : m_held( new T( rhs ) )
        {}

        virtual ~holder_T( )
        {}

        virtual shared_ptr< holder_i > clone()
        { return new holder_T< T >( *m_held ); }

    private:

        shared_ptr< T > m_held;
    };
```

Then what `any` will do is holding a pointer of type `holder_i*`, and `any_cast` will turn it back into `holder_T< T >*` using the RTTI feature of C++:

```
class any
{
public:

    template< typename T > any( const T& rhs )
        : m_holder( new holder_T< T >( rhs )
    {
    }

    any( const any& rhs )
        : m_holder( rhs.m_holder->clone() )
    {
    }

    virtual ~any()
    {
    }

private:

   shared_ptr< holder_i > m_holder;

   friend template< typename T >
   T any_cast( const any& ins )
   {
       shared_ptr<holder_T<T> > ptr;
       ptr = dynamic_ptr_cast<holder_T<T> >
           (
               ins.m_holder
           );

       if( ptr == NULL )
       {
           throw bad_alloc;
       }

       return *(ptr->held);
   }
```

The actually code of any is a little bit different from the above, but the idea is the same.

1.3.2 `mem_fn` and `bind`

Class `mem_fn` is a template function from boost which turns a member function into a normal function. For example, if we have the following class A defined:

```
class A
{
public:

    void set( int v)
    {
        m_value = v;
    }

    int get()
    {
        return m_value;
    }

private:

    int m_value;
}
```

Then the following code:

```
    A a;
    a.set( 0 );
```

can be rewritten to the following using `mem_fn`:

```
    A a;
    mem_fn(&A::set)( a, 0 )
```

Meanwhile, bind is also from boost which is used to feed dummy argument to functions to generate new functions with less arguments. Still used the above example, support we have the function `set_a` defined as following:

```
   void set(A& a, int v)
   {
       a.set( v );
   }
```

Then the following code:

```
  A a;
  set_value(a, 0);
```

can be changed into the following:

```
  bind( set_value, _1, 0)(a);
```

1.3.3 `boost::function`

The template `boost::function` is boost's replacement for function ptr. Say you have the following function defined:

```
void func(int a1, double a2, float a3, vector<int>& out);
```

To store its pointer, you would need to use the following code:

```
void (*funcptr)(int, double, float, vector<int>& );

funcptr = &func;

vector<int> iv;
(*funcptr)( 1, 1.0, 1.0f, iv);
```

using `boost::function`, you can use the following code:

```
boost::function< void (int, double, float, vector<int>&) > func2 = func;

vector<int> iv;
func2(1, 1.0, 1.0f, iv);
```

As can be seen, the `boost::function` code is more clean. Moreover, `boost::function` can be used for functor, as the following exmaple shown:

```
struct functor_t
{
  void operator()(int, double, float, vector<int>&)
  {
    ....

  }
};

boost::function< void (int, double, float, vector<int>&) > func3 = functor;

vector<int> iv;

func3(1, 1.0, 1.0f, iv);
```

`boost::function` has a lot of advantages over function pointer, please refer to its document for all of them.

1.3.4 `boost::functor`

`boost::functor` includes boost's own implementation of some usual functors, such as `ptr_fun`, `not1`, etc. They are for many reasons better than STL's implementation.

1.3.5 Other libraries

A lot of other components of boost has been used in mort, but we do not have enougt time to explain them detailly, just list them here.

smart_pointer contains several kinds of pointers, one of them we have discussed is `shared_ptr`, others include `scoped_ptr`, etc.

mpl is a Meta Programing Language module, we will introduce some basic concept of it in the future, a detailed description can be found in reference.

str_algo includes a lot of string algorithms such as find, replace, copy, split.
Test is a software test framework, all the test case under directory mortest is setup using this framework.
Filesystem is multi-platform C++ library handling directory, file paths problems.
Python is used to setup a python binding for leap.

Chapter 2

Introduction to MORT

MORT (Molecular Objects and Relevent Objects) is a C++ library for all kinds of chemical operations,it is used by gleap as a foundation, but is not limited in this usage. User can write a lot of powerful programs using it. In this chapter, we will generally introduce the classes and functions defined in mort, and we will show you some examples about how to use MORT in the next chacter.

2.1 molecule

Molecule is the center concept of MORT, and there is a class to represent it, which is `molecule_t`.

The following code shows the declaration of a molecule, which is fairely simple:

```
molecule_t m;
```

After you declare a molecule, you can perform all kinds of operations on it. The simplest operation is to access a parameter of molecule, (Note the term "parameter" is of the same meaning as "property" and "attribute", we choose to use it instead of the other two because the other two terms has been used by some programming languages (Jave, Python) to denote their built-in features.) which is acturaly not as simple as you think. The complexity comes from the following aspects.

Firstly, each parameter has two parts: name (or "key") and value. Key is usually a string but for some frequently used parameters such as "name" and "type", it would make sense that we can use an integer ID to retrieve to improve efficiency.

Secondly, the value of parameters can be of any type, which requires a general mechanism to store variable of any type and later retrieve it back. As you may has realized, the boost module `boost::any` introduced in the first chapter is a perfect candidate for this job. The only problem `boost::any` will have is efficiency, it would be very unefficient to use `boost::any` to store primary data types like `int` and `double`.

Thirdly, there should be some easy way to test if a parameter has been set or not.

Finally, there should be some easy way to delete a parameter.

The parameter access mechanism of MORT is established according to these requirement, and can be described as the following:

1. Parameters are stored in an integer indexed map. It is still possible to retrieve paramter through its name (which is a string), although the name is translated to an integer hash ID using MORT's hash function during the retrieving. (We will introduce more detail about MORT's hash function in the next chapter, right now you just need to know it is a 1 to 1 mechanism.) For some frequently used paremters, their hash ID has been pre-calculated and save in header file *mortsrc/common/hashcode.hpp*, so that they can be retrieved using ID to save the effort of translation. The following table shows these parameters:

Name	Hash ID	Data type	Description
name	NAME	string	molecular name
type	TYPE	string	molecular type, could be monomer, protein, nucleic acid or other
solvate	SOLVATE	int	solvation type, could be BOX, CAP, OCT or SHELL
box	BOX	numvec	solvation box info, should be (xlen, ylen, zlen, theta
cap	CAP	numvec	solvation cap info, should be (center_x, center_y, center_z, r)
shell	SHELL	double	solvation shell thickness

2. Totally 4 types of member fucntions are provided. They are called setter, getter, tester and deleter respectively, and the function name will be `set_x`, `get_x`, `has_x`, and `del_x`. Note here 'x' is parameter type denotion, it could be one of 'i', 'd', 's', 'v', 'a', see next pragraph for detail.

3. Parameters are classified into five categories according to their data types. These five categories are: integer (denoted as 'i'), double (denoted as 'd'), string (denoted as 's'), numvec (numeric vector is mort's data type, denoted as 'v'), and all other data types (denoted as 'a', stored as `boost::any`). Each category has its own setter, getter, tester and deleter functions. For example, to set an integer parameter you should use function `set_i`, while to get a double parameter you should use function `get_d`, etc.

4. setter function takes two arguments, the first one is parameter identifier which as mentioned before could be a string or a 64-bits integer, the second one is paramter value whose data type varies according to the denotion.

5. There exist two sets of getter functions. The first set takes only one argument, the parameter identifier (could be either a string or a hashid), and return the reference of the parameter, if the requested parameter does not exist, it will throw an exception of type `parm_error`. The second set, which is sometimes called "safe getter" takes two arguments, the first one is still the parameter identifier, the second one is reference to data value. If the requested parameter does exist, getter will change the value of second argument to parameter value and return a boolean value `true`, otherwise it will return boolean value `false`, and the second argument remains unchanged.

6. Tester functions takes only one argument, the parameter identifier and return a boolean value indicating if the requested parameter exists.

7. Deleter functions takes the parameter identifier and delete it. return value could be `true` indicating success or `false` indicating failure, i.e. the parameter does not exist.

The following code show to how these functions acturally works:

```
molecule_t m;
m.set_s( "name", "methane" ); // string as identifier
std::cout << m.get_s(NAME) << std::endl; // ID as identifier

std::cout << "Does␣m␣has␣a␣name?" <<   m.has_s("name") << std::endl;
```

Code above shows how to access molecular's parameter through member functions of class `molecule_t` Class `molecule_t` has a lot of other member functions involving atom, bond and residue manipulation, but we will have to introduce the concept of "molecule object" first.

2.2 molecular object

Molecular object is another important concept of MORT. Currently there are six kinds of molecular objects: each has been assigned a 4-character alias and a unique hash code:

Description	Alias	Hash code
bond	bond	BOND
angle	angl	ANGL
torsion	tors	TORS
out-of-plane stretch (improper torison)	oops	OOPS
torsion-torsion	tor2	TOR2
PI-torsion	ptor	PTOR

As you may have noticed all these names are four characters long, this is not a coincidence but a selected result, please keep this rule when you are trying to introduce new kind of molecule object to MORT, we will introduce how to do that in the next chapter.

2.2.1 To create a molecule object

It should be noted that molecular object is not a physical existence. i.e. it is always associated with a molecule. When you create a molecular object or make a change to it, you actually made changes to the molecule. Consequently, we cannot retrieve a standalone molecular object. All operations to a molecular object must be made through "molecular object reference" (represented by class `moref_t`) or "molecular object iterator" (represented by class `moiter_t`).

To create a molecule object, you can use member function of class `molecult_t`. Class `molecule_t` has a series of fucntions for molecular object creation, function name is like `create_xxxx` in which "xxxx" should molecule object alias. These function takes no argument and return a "molecular object reference"

2.2.2 To set parameter of a molecule object

After you got a molecular object reference, you can perform all kinds of operation on it. The basic operation would be setting and getting its parameter. The parameter accessing mechanism works exactly like the one for `molecule_t`. i.e. you can access a parameter through its name or ID; there are five categories of parameters, denoted as 'i', 'd', 's', 'v', 'a'; and there are setter, getter, tester and deleter functions for each parameter. The following table summerized some frequently used parameter of all kinds of molecule objects

Name	MOs	description
aapos	resd	amino acid position, option: NTERM, OTHER, CTERM, ALONE
brnr	atom	Born radii for GB
depth	atom	Potential well depth, one of the two VDW parameters
element	atom	atomic number
equil	bond, angl, tors	equivelant value of force field
fchg	atom	formal charge
force	bond, angl, tors	force constant
gbfs	atom	GB factor of screening
hybrid	atom	hybridization type, could be SP1, SP2, SP3
lescopy	atom	LES copy number
lestype	atom	LES type id
name	atom,resd	name
order	bond	bond order
period	tors	period
position	atom	position or coordinates
rstar	atom	VDW radii star, another VDW parameter

the following code showing constructing a methane:

```
molecule_t m;
moref_t c0 = m.create_atom();
moref_t h1 = m.create_atom();
moref_t h2 = m.create_atom();
moref_t h3 = m.create_atom();
moref_t h4 = m.create_atom();

c0.set_s( NAME, "C0" );
h1.set_s( NAME, "H1" );
h2.set_s( NAME, "H2" );
h3.set_s( NAME, "H3" );
h4.set_s( NAME, "H4" );

moref_t b0 = m.create_bond();
moref_t b1 = m.create_bond();
moref_t b2 = m.create_bond();
moref_t b3 = m.create_bond();

b0.set_i( ORDER, 1 );
```

```
b1.set_i( ORDER, 1 );
b2.set_i( ORDER, 1 );
b3.set_i( ORDER, 1 );
```

2.2.3 Making relations between molecular objects

The example code showing in the last subsection is not sufficient to create a methane, since there is no necessary relation between atoms and bonds, i.e. that atom "c0" should be connected to other atoms.

"Relation" is one thing that MORT different from some software package such as NAB and leap which uses hierachy framework. In hierarchy mode, a molecule owns some residues, and each residue owns some atoms, and to iterate on atoms, one need to first iterate on residues. This framework has two problems, firstly it can not handle monomer very easily; secondly, it is very hard to extend, say if one want to add another hierachy "chain", which owes residues, then all the iteraton code on atom need to be rewritten to add iteration on chain.

The framework MORT used is so-called "relation" framework. In this framework, molecule owns every molecular directly, and residues does not own atoms, they are just "related" (or in other word "connected".) So do other molecular objects. i.e. atoms does not own their neighbours they are just connected to them, meanwhile bonds, angles and torsions does not own the atoms, they are just related to them. To iterate on atoms of a molecule, one does not need to go through residues, thus molecule is not required to has residues. If you want introduce new molecular object "chain", and want it to be connected to some residues you can just create it and make connection. As for the old code, they do not need to be changed since they know nothing about chain. In short, relations you do not know will not hurt you.

class `moref_t` has three member functions which can be used to create delete and test relations between molecular objects respectively: `connect()`, `disconnect()`, and `is_connected_to()`. As an example, the following code should be added to finish our example about creating a methane:

```
  // creating the 1st bond
  c0.connect(h1);
  h1.connect(c0);
  c0.connect(b0);
  b0.connect(c0);
  h1.connect(b0);
  b0.connect(h1);

  // creating the 2nd bond
  c0.connect(h2);
  h2.connect(c0);
  c0.connect(b1);
  b1.connect(c0);
  h2.connect(b1);
  b1.connect(h2);

  // creating the 3rd bond
  c0.connect(h3);
  h3.connect(c0);
  c0.connect(b2);
  b2.connect(c0);
  b2.connect(h3);
  h3.connect(b2);

  // creating the 4th bond
  c0.connect(h4);
  h4.connect(c0);
  c0.connect(b3);
  b3.connect(c0);
  h4.connect(b3);
  b3.connect(h4);
```

as you can see, it is tedious to create bond. To create a single bond, you need call `connect` 6 times. Fortunatly, MORT provides functions to create bond, angl, tors, and make right connection for you. as shown in the following:

```
moref_t create_bond( moref_t& a0, moref_t& a1 );
moref_t create_angl( moref_t& a0, moref_t& a1, moref_t& a2 );
moref_t create_tors( moref_t& a0, moref_t& a1, moref_t& a2, moref_t& a3 );
...
```

thus the overall code to construct a methane can be written as the following:

```
molecule_t m;
moref_t c0 = m.create_atom();
moref_t h1 = m.create_atom();
moref_t h2 = m.create_atom();
moref_t h3 = m.create_atom();
moref_t h4 = m.create_atom();

c0.set_s( NAME, "C0" );
h1.set_s( NAME, "H1" );
h2.set_s( NAME, "H2" );
h3.set_s( NAME, "H3" );
h4.set_s( NAME, "H4" );

c0.set_i( ELEMENT, 6 );
h1.set_i( ELEMENT, 1 );
h2.set_i( ELEMENT, 1 );
h3.set_i( ELEMENT, 1 );
h4.set_i( ELEMENT, 1 );

moref_t b0 = create_bond( c0, h1 );
moref_t b1 = create_bond( c0, h2 );
moref_t b2 = create_bond( c0, h3 );
moref_t b3 = create_bond( c0, h4 );

b0.set_i( ORDER, 1 );
b1.set_i( ORDER, 1 );
b2.set_i( ORDER, 1 );
b3.set_i( ORDER, 1 );
```

2.2.4 Iterating on molecule objects

The iteration on molecular objects is very important, there are two methods to do this in MORT, through molecule object iterator (class `moiter_t`) or through molecule object range (class `morange_t`).

Class `moiter_t` works just like other iterators, dereferencing it will give you a molecular object reference. Class `molecule_t` has several member function returning molecular object iterator, the format is like `xxxx_begin()` and `xxxx_end()`, here `xxxx` could be any of molecular object names such as "atom" and "bond". Class `moref_t` also has member functions returning iterators of its related molecule object. The function name is in the format `related_xxxx_begin()`, `related_xxxx_end()`, again `xxxx` could be any of "atom", "bond", "resd", etc.

The following code illustrates the iteration on atoms on a molecule and a residue and print out its name:

```
  molecule_t m;

  ... ...
```

```
moiter_t ai = m.atom_begin();
for( ; ai != m.atom_end(); ++ai )
{
    std::cout << "atom␣name:" << ai->get_s(NAME) << std::endl;
}

moref_t r = m.create_resd();

....

moiter_t aj = r.related_atom_begin();
for( ; ai != r.related_atom_end(); ++aj )
{
    std::cout << "atom␣name:" << aj->get_s(NAME) << std::endl;
}
```

Molecular object range works just like an array, and is good for random access. Class `molecule_t` has member functions `xxxxs()` returning molecular object range, and class `moref_t` has similar member function `related_xxxxs()`. The following code re-implement the above iteration using molecular object range:

```
molecule_t m;

......

morange_t atms1 = m.atoms();
for(int i=0; i < atms1.size(); ++i )
{
    std::cout << "atom␣name:␣" << atms1[i].get_s(NAME) << std::endl;
}

moref_t r = m.create_resd();

....

morange_t atms2 = r.related_atoms();
for(int i=0; i < atms2.size(); ++i )
{
    std::cout << "atom␣name:␣" << atms2[i].get_s(NAME) << std::endl;
}
```

2.3 Utility Classes

Other than those we have introduced, there are several important utility classes in MORT.

2.3.1 Entity

As we mentioned before, unlike other OO language, C++ does not have a single-rooted inhertience system, which is inconvinient sometimes. To overcome this problem, we introduced entity `entity_t` which is the root class for mort, i.e. every physical object should inherit from it, such as `molecule_t` and `database_t` (will be introduced later) etc. As for molecular object `moref_t`, it is not physically existing thus not inheriting from entity.

The purpose of introducing entity is not limitted to provide a root class, class `entity_t` has a full set of member function for parameter access. In a sense, those parm setters, getters of class `molecule_t` is in fact inherited from class `entity_t`.

2.3.2 Database

One advantage of having a root class is that one can now implement the composite design pattern. Composite is a way to organize many different things (molecule, atom vector, name map, etc) into one single container: first you implement them such that they all inherit from the root class, then the composite class (in this case class `database_t` will contain an array of pointer to the root class, these pointers can be casted back to their original data type by `dynamic_cast` or `dynamic_pointer_cast` if you use `shared_ptr` which we highly recommend. Moreover, one can have the composite class inherit from the root class too, which will allow user to build up a hierarchical storage class.

If you are interested in the composite design patterns or other design patterns, we highly recommend you to read the classical design patter book *Design patterns: Elements of Resuable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

The following are member functions of class `database_t`:

```
bool has( const string& name ) const;

void set(const string& name, const entity_ptr& pmol );

void add(const string& name, const entity_ptr& pmol );

bool remove(const string& name);

entity_ptr get( const string& name ) const;

molecule_ptr get_mol(const string& name) const;

database_ptr get_mdb(const string& name) const;

atomvec_ptr get_avec(const string& name) const;

bondvec_ptr get_bvec(const string& name) const;

namemap_ptr get_nmap(const string& name) const;

iterator begin();

iterator end();
```

Most of these functions are straightforward. Basically `database_t` works in a name value pair way. One can store an entity by `set()` and retrieve it later by *get*() as for `get_mol()` it does the type cast for you.

One might find it confusing that there are function `add()` and `set()` at the same time. What is the difference between them? while `set()` will go over existing records and update the value if the given name already existed, `add()` simply add new record to the end of the record set. In other word, if you want `database_t` act like a map, use `set()`, if you want `database_t` act like an array, use `add()`.

2.4 Atom Vector

You may have noticed in the previous section that class `atomvec_t` appeared many times. This is MORT's data type to represent a vector of atoms.

class `atomvec_t` inherit both from `entity_t` and from `vector<moref_t>`. Besides it has the following useful member functions:

```
void push_atom( const moref_t& mo );

atomvec_t::iterator find( const moref_t& mo );
```

The function `push_atom` takes a molecular object, if it is not an atom, all its related atoms will be pushed into it.

The function `find` search for the given atom in the vector, if not found return `end()`, it is just a wrapper of standard algorithm `find`.

2.5 Utilities functions

Along with all the objects, MORT has a handful of utility functions:

1. File format handling functions. MORT now can understand the following standard file format: MDL's sdf, Tripos mol2, PDB, SMILES. Most importantly MORT understand AMBER's file format prep (read only), off and prmtop (write only).

2. Substructure search and atom typing. MORT has very efficient algorithm to do substructure matching. It can also do atom typing based on SMARTS pattern language.

3. Molecular preparation. MORT has the ability to solvate molecule in different shapes: cap, box, octrahedraon, shell, etc. MORT can also add counter ions according to electrostatic potential.

4. Interfacing with SFF. MORT provides an interface to SFF to allow user to run energy calculation, minimization and MD simulation from a raw PDB file.

In the next chapter we will demonstrate how to use these functions by some examples.

Chapter 3

Examples showing how to use MORT

In this chapter, we will demonstrate the usage of MORT's utility functions by developing some useful programs.

3.1 formatter

In this section, we are going to develop a file format conversion tool named "formater", and you are getting to know more about class `molecule_t`, and the molecule file IO interface of MORT.

Format conversion is important in molecular simulations works, we do it on daily bases. In this section, we will show how to use mort to write a format conversion tool.

Currently mort supports the following file formats: mdl's sdf format, tripos's mol2 format, brookhaven's pdb format, daylight's SMILES, tinker's xyz format, amber's off format, prmtop format(only for writing) and inpcrd format (only for writing). For each format, there is a hashcode defined in file *mortsrc/common/hashcode.hpp*, like the following:

```
namespace mort
{
    ....;

    ....;
}
```

Two things should be emphasized here are:

1. all the classes functions of mort and declared inside namespace mort.

Mort's main interface for molecular file IO made up of the following functions declared in file *gleap/mortsrc/format.hpp* as the following:

```
namespace mort
{
    .....

void load_mol(istream& is,molecule_t& mol,int format);

void load_mdb(istream& is,database_t& mol,int format );

void save_mol(ostream& is,const molecule_t& mol,
              int format );

void save_mdb(ostream& is,const database_t& mol,
              int format );

void load_mol(const string& file,molecule_t& mol,
              int format=UNKNOWN);

void load_mdb(const string& file,database_t& mol,
              int format=UNKNOWN);

void save_mol(const string& file,const molecule_t& mol,
              int format=UNKNOWN);

void save_mdb(const string& file,const database_t& mol,
              int format=UNKNOWN);

    ....
}
```

The prototypes are almost self explanary. Noted here several things:

1. when calling `load_mol` or `save_mol` with a file name but no format ID, the format ID will be decided by the file extention.

2. An exception of type `std::exception` will be thown when encoutering an error and unable to get through it, for example, can't open input file or output file or unknown format type, etc.

The following code shows the first version of `formater` which takes the first argument to be input file name,and the second argument to be output file name:

```
#include <iostream>
#include <object.hpp>
#include <format.hpp>

using namespace std;

using namespace mort;

int main( int argc, char** argv )
{
  if( argc != 3 )
  {
    cerr << ``usage: formater input output'';
    cerr << std::endl;
    return -1;
  }

  try
  {
    molecule_t mol;
    load_mol( argv[1], mol );
    save_mol( argv[2], mol );
    return 0;
  }
  catch( std::exception& e )
  {
    std::cerr << e.what() << std::endl;
    return -1;
  }
}
```

Noted here the statement `using namespace mort;` export all the mort functions to public, you would always like to do that unless you have a name conflicting.

This version of formater is trivial, and has only limited functionalities. Firstly, when loading PDB files it will try to makes bonds between atoms close enough, sometimes it will causes error; Secondly, pdb files does not have charge information, so when outputing to mol2 format, the column of atomic charges are all 0.0; Thirdly, files in sdf format or mol2 format could contain multiple molecule, for such a file we want the output also to be a database.

To solve these problems, we revised it a little bit, makes another version. These version has the following improvment:

1. when reading a pdb file, it will try to read some pre-defined amino acid templates stored under directory *$AMBERHOME/dat/leap/lib/* which contain the charge and connection information, and use these information to build up atoms.

2. when reading a sdf file or a mol2 file, it will try to reading all molecules and store them in a `database_t`, and save all molecules

Below is the source code of the new version:

```
#include <sstream>
#include <fstream>
#include <object.hpp>
#include <format.hpp>

using namespace std;

using namespace mort;

int main( int argc, char** argv )
{
  if( argc != 3 )
  {
    cerr << ``usage: formater input output'' << std::endl;
    return -1;
  }

  try
  {
    database_t db;

    int format_in = get_fmt( argv[1] );

    if( format_in == PDB )
    {
      const char* amberhome = getenv( ``AMBERHOME'' );

      std::cout << ``finding amber home '';
      std::cout << amberhome << std::endl;

      if( amberhome != NULL )
      {

        load_mdb( amberhome +
          string(``/dat/leap/lib/all_amino03.lib''),
          db );
        load_mdb( amberhome +
          string(``/dat/leap/lib/all_aminoct94.lib''),
          db );
        load_mdb( amberhome +
          string(``/dat/leap/lib/all_aminont94.lib''),
          db );
      }
    }

    molecule_t mol;
    ifstream is( argv[1] );
    ofstream os( argv[2] );

    if( !is || !os )
    {
      string bad_file = (!is) ?argv[1]:argv[2];
      string bad_oper = (!is) ?``read'':``written'';
      throw logic_error(``can not open file ''+
                          bad_file+
                          `` for '' +
                          bad_oper );
    }

    int format_out= get_fmt( argv[2] );

    load_mol( is, mol, format_in);
    if( format_in==PDB)
    {
        mdlize_mdb(mol, db);
    }

    save_mol( os, mol, format_out);

    if( is_dbfmt( format_in ) )
    {
      int nmol = 1;

      while( is )
      {
        molecule_t tmp;
        load_mol( is, tmp, format_in );

        if( atom_number( tmp ) > 0 )
        {
          if( is_dbfmt( format_out ) )
          {
            save_mol( os, tmp, format_out );
          }
          else
          {
            ostringstream filename;
            filename << argv[2] << ``.'' << nmol;
            save_mol( filename.str(), tmp, format_out );
          }
        }

        nmol++;
      }

    }

    return 0;
  }
  catch( std::exception& e )
  {
    std::cerr << e.what() << std::endl;
    return -1;
  }
```

```
}
```

As you may have noticed, we added some special handling for PDB format. First, libraries contain models were loaded; Second, a special function `mdlize_mdb()` was called which "modelize" a the raw molecule (This is a word we invented which means building up a molecule based on model molecules)

The code is located in the directory *gleap/example/mort/formater*, along with a Makefile like the following:

```
all : formater.v1 formater.v2

formater.v1 : main_1.o
        g++ -o formater.v1 main_1.o -lmort -L../../../mortsrc

formater.v2 : main_2.o
        g++ -o formater.v2 main_2.o -lmort -L../../../mortsrc

main_1.o : main_1.cpp
        g++ -c main_1.cpp -I../../../mortsrc

main_2.o : main_2.cpp
        g++ -c main_2.cpp -I../../../mortsrc
```

3.2 impose

In this section, we are going develop a program called "impose", which for a molecule can impose one internal coordinate (distance, angle or torsion) without affecting others. In this example, you are getting to practice with some important components of mort: atom, bond, residue and atomvec.

For biological systems, sometimes it is important to change only one internal coordinate to observe the influence on energy of conformation change. This is the motivation to write such a program.

The key point of impose is to find out all the atoms connected to the target atom. Here by connected, we meant not only the atoms bonded to the target atom, but also atoms bonded to its neighbours, and neighbours' neighbours, which requires a recrusive function to do that. The following function `list_nbrs` is designed for this goal:

```
  void list_nbrs(const moref_t& curt,const moref_t& prev,
                 const moref_t& root,atomvec_t& list)
  {
    list.push_back( curt );

    iterator_t next = curt.related_atom_begin();

    for( ; next != curt.related_atom_end(); ++next )
    {
      if( *next == prev )
      {
        continue;
      }

      if( *next == root )
      {
        throw logic_error( ``ring found!'' );
      }

      if( list.fin(*next)==list.end() )
      {
        list_nbrs(next,curt,root,list);
      }
    }
  }
```

We understand you might have a lot of questions about the above code, let's go over the function line by line starting from the prototype:

```
void list_nbrs(const moref_t& curt,const moref_t& prev,
               const moref_t& root, atomvec_t& list )
```

As the varibale name, `curt` stands for current, `prev` stands for previous. `list\_nbrs` is a recrusive function, and calls itself later with different argument, so we think it is helpful that variable named in this way. `moreft_t` as we have explained is molecular object reference here it is referring to an atom.

Variable `list` is of type `atomvec_t`, which as we explained before is atom's container. `list` will contain the result.

The next statement

```
  list.push_back( curt );
```

simply push the current atom to the end of `list` (you may recall now that `atomvec_t` inherit directly from `vector<moref_t>`, and `push_back` is a member function of its parent type.)

OK, here comes the next statemement:

```
moiter_t next = curt.related_atom_begin();
for( ; next != nbr_end( curt ); ++next )
```

This as we have explained in the last chapter, is one of the two typical ways to iterate on an atoms' neighboring atoms. The other way is through molecular object range (class `morange_t`).

The next part is:

```
if(*next==prev)
{
  continue;
}

if(*next==root)
{
  throw logic_error( ``ring found!'' );
}
```

The statment `(*next==prev)` involves two important functions. Firstly, statement `*next` calls the member function `operator*()` of class `moiter_t` and returns a "molecule object reference" the iterator is pointing to. Then, the function `bool operator==(const pointer_t& lhs, const pointer_t& rhs)` was called which compare the absolute ID of the two atoms and return the result.

the atom `root` is the starting point of the whole travel, thus if `next` equals to `root`, we have a ring in the system and we are unable to establish a clean list of related atom, so we throw out an exception to notify the caller.

The next part is:

```
if( list.find(*next)==list.end() )
{
  list.push_back(next);
  list_nbrs(next,curt,root,list);
}
```

Here `find` performs a linear search of target on the whole atom vector. If it returns list.end(), means we can not find `next` in the atom vector `list` and indicating it has never been visited before, should be put in `list`, and be visited on the next round.

Note it is a typical DFT (Depth First Travel) implementation. It is also possible to re-write the function using BFT (Breath First Travel) strategy. We will leave it as an execise for the user. The answer will be given in the following sections.

Now we have finished the hardest part, here listed is the implementation of function `impose_dist`, which impose the distance between two atoms:

```
void impose_dist( const atomvec_t& atoms, double dist )
{
  assert( atoms.size() == 2 );

  atomvec_t nbr_0, nbr_1;
  list_nbrs( atoms[0], atoms[1], atoms[1], nbr_0 );
  list_nbrs( atoms[1], atoms[0], atoms[0], nbr_1 );

  numvec v0 = atoms[0].get_v(POSITION);
  numvec v1 = atoms[1].get_v(POSITION);

  double curt = mort::dist( v0, v1 );

  if( set_0.size() <= set_1.size() )
  {
    target = &set_0;
    offset = (v1-v0) * ( 1 - dist / curt );
  }
  else
  {
    target = &set_1;
    offset = (v0-v1) * ( 1 - dist / curt );
  }

  for( int i=0; i < target->size(); ++i )
  {
    numvec pos = target->at(i).get_v(POSITION);
    pos += offset;
    target->at(i).set_v(POSITION, pos);
  }
}
```

The code is almost self-explained, we just put some reminder here:

1. `numvec` is the numeric vector type of MORT.

2. function call `get_v(POSITION)`, get the atom's position, which assume atom already has position, if not, it will throw an exception.

3. The function `dist` is provided by mort to measure the distance between two numeric vectors.

4. The algorithm list atom's neighbors from both sides, and only translate the side with less atoms.

The implementation of function `impose_angl` and `impose_tors` is similar and can be found in file *gleap/example/impose/impos.cpp*.

Here comes the entry point of impose:

```
int main( int argc, char** argv )
{
  if( argc != 4 )
  {
    cerr << ``usage: impose input output constrain'';
```

```
      cerr << std::endl;
      return -1;
    }

    try
    {
      molecule_t mol;
      load_mol( argv[1], mol );

      double value;
      vector< string > names;
      transfer( argv[2], names, value );

      iterator_t ri = mol.resd_begin();
      for( ; ri != mol.resd_end(); ++ri )
      {
        atomvec_t atoms;
        for( int i=0; i < cons.names.size(); ++i )
        {
          atoms.push_back( ri->get_related_atom(names[i]) );
        }

        if( atoms.size() == 2 )
        {
          impose_dist( atoms, value );
        }
        else if( atoms.size() == 3 )
        {
          impose_angl( atoms, value );
        }
        else if( atoms.size() == 4 )
        {
          impose_tors( atoms, value );
        }
        else
        {
          throw logic_error( ``wrong constraint''
                             + string( argv[2] )
                           );
        }
      }

      write_mol( argv[3], mol, MOL2 );

      return 0;
    }
    catch( std::exception& e )
    {
      std::cerr << e.what() << std::endl;
      return -1;
    }
  }
```

In this code, we were iterating on each residue of the molecule, look for atoms with the specified names, then impose them. Here we used an iterator `ri` to iterate on all residues of a molecule, function call `ri->get_related_atom(names[i])` returns the atom with given name.

The full code can be found in file *gleap/example/mort/impose/impose.cpp*, with a Makefile and some inputs you can play with.

3.3 subfinder

In this section, we illustrate how to use mort to build up a substructure finder. We are going to introduce SMILES and SMARTS language, and mort's interface for interpreting them and for finding substructure.

In pharmacutial researchs, sometime we need to look for a certain substructures in a molecule, this substructure could be a benzene ring, or more compilcated functional group. MORT has some built-in functions for this purpose. Firstly, mort have the ability to interpret SMILES or SMARTS, which allow you to specify a substrcutre using a string; Secondly, mort has functions to find substructure in molecule.

SMILES is a language proposed by Daylight Inc. for the description of a chemical structure. The rules of SMILES are as simple as the following:

1. use atomic symbol to represent atoms, i.e. 'C' for a carbon, 'n' for a nitrogen, lower cased atomic symbols for aromatic atoms.

2. for bond representation, '-' for single bond, '=' for double bond, '#' for triple bond, and ':' for aromatic bond.

3. use parenthesis to represent branch structure, for example, CC(C)C stand for 2-methyl-butane.

4. use number to note for a ring. For example, C1CCCCC1 stand for a hexane.

MORT's interface to interpert SMILES string is `read\_smiles`, the prototype is like:

```
void read_smiles(const string& smiles,molecule_t& mol);
```

and the interface to find substructure is `find\_subst` and `has\_subst`, here are the prototypes:

```
bool find_subst(const molecule_t& mol,
                const molecule_t& sub,
                atomvec_t& path);

bool has_subst(const molecule_t& mol,
               const molecule_t& sub);
```

we list here the code of "subfinder". The program takes two arguments, the first one is a molecular file in SDF format (could either be a single file or a database), the second one is a SMILES string specifying a substructure.

```
  int main( int argc, char** argv )
  {
    if( argv != 3 )
    {
      cerr << ``usage: subfinder moldb smiles_string'';
      cerr << std::endl;
      return -1;
    }

    try
    {
      molecule_t sub;
      read_smiles( argv[2], sub );

      ifstream is( argv[1] );
      while( is )
      {
        molecule_t mol;
        read_sdf( is, mol );

        atomvec_t found = find_subst(mol,sub);

        if( found != NULL )
        {
          std::cout << ``Found substr in molecule '';
          std::cout << mol.get_s(NAME) << std::endl;
          std::cout << ``Starting from atom ''
          std::cout << found[ 0 ].absid()
          std::cout << std::endl;
        }
      }

      return 0;
    }
    catch( std::exception& e )
    {
      std::cout << e.what() << std::endl;
      return -2;
    }
  }
```

This version of subfinder is nice and powerful, but it would be even better if we can have the some more functionalities. In the real world of pharmacore screening, we seldom know exactly what kind of substructures we are looking for. Most of the time, we only have some rough ideas about how the candidate should look like. For example, at some point we want the atom to be any hydrogen bond donor, which could either be an oxgen or a nitrogen, but SMILES won't allow you to be arbitary on atoms. Similarily, sometimes we want have some restrictions on atoms, like "it must be of SP2 hybrid", "it has at least one hydrogen connected", again, SMILES won't allow that, nether does any other file format we know.

Fortunately, there is yet another pattern language called SMARTS can be used for this purpose which is also supported by mort. The grammer SMARTS is simple. First of all, all the SMILES syntax still works for SMARTS, i.e. all SMILES strings are valid SMARTS target. What is new about SMARTS is the introduction of "atomic pattern". In a SMARTS string, content between '[' and ']' are regard as "atomic pattern", which has following rules:

1. Several descriptors have been introduced for the better description of chemical environment, including 'X'(for neighbor), 'V'(for valence), 'R'(for ring), 'r'(for aromatic ring), '$\hat{}$' for hybrid. For example, atomic pattern [X4] means an atom with 4 neighbors, while [R6] means an atom in a six membered ring, and [$\hat{3}$] is a SP3 hybrid atom.

2. direct connected patterns are in "and" relation, comma separated patterns are in "or" relation,colon seperated patterns are in 'and' relation, and the preference is *direct connection* $>$ *comma* $>$ *colon*.

Thus, [C$\hat{2}$] means an atom which "is carbon in SP2 hybrid", wile [O,N;R6] means an atom "could be either an oxygen or nitrogen, but must be in a six membered ring".

Change subfinder to use SMARTS is easy, just use function `read\_smarts` to replace function `read\_smiles` and we are done.

Here we would want to spend sometime the explain the implementation details of `read\_smarts` and `find\_subst`, please feel free to skip this part if you are not interested.

The difference between `read\_smarts` and `read\_smiles` is that `read\_smiles` read and set atomic parameter "element" while `read\_smarts` interpret "atomic pattern" into an object "pattern" and store it as an atomic parameter. When comparing substructural atom and molecular atom, `find\_subst` will first test the existence of parameter "pattern" in the substructure atom, if it does exist, get the pattern and use it to match the molecular atom, otherwise compare the parameter "element" of the two atoms.

As you may have noticed, subfinder could make a solid foundation for a pharmacore software if combined with conformation change (some part of it has been described in impose) and geometry constraint.

3.4 hlogs

In this section, we will illustrate the development of a soluability predictive tool hlogs (Hou's logS). We will illustrate how to use the atom typing utility of mort and ring detection function.

Soluability (logS) is a very important property of leading compound, since it has a big impact on drug's absorption and distribution in human body. Here we will implement the method and parameters proposed by Tingjun Hou et. al. for soluability prediction(Journal of Chemical Information and Computer Science 44(1): 266-275, 2004). This method is one of those atomic additive approach, which consider logS being the sum of atomic contribution. In Hou's method, he classified atoms into different types according to their environment. Atoms of same type contribute equally to logS, and the empirical value of each atom type's contribution to logS were obtained by fitting to the experimental results.

The procedure of "classifying atoms according to their environment" is called "atom typing". As you may have noticed, the SMARTS language we introduced in last section is ideal for describing "environment". To do this job, we need define a set of environment (in SMARTS), and get the parameter for each of them.Then we can do logS prediction. In fact mort has encapulsed this whole procedure into a single class `typing\_rule`, which reads a typing rule from a input stream, and assign atom type according to this rule.

The following is a piece of the typing rule used by hlogs:

```
# typing rule for logS

  pattern                           typeid
  [CX4;H4]                               1
  [CX4;H3]                               1
  [CX4;H3][#6]                           2
  [CX4;H3][CX4,c,F,Cl,Br,I]              3
  [CX4;H3][CX3,c,F,Cl,Br,I]=[#8,#7]      4
  [CX4;H3][CX4,c,F,Cl,Br,I]~[#8,#7]      5
```

The format of a typing rule file is simple:

1. empty lines and lines starting with a sharp('#') are considered as comments and will be ignored;

2. first non-comment line is considering as "title line", phrases seperated by spaces are names of each column, and the coming lines should be inputed in the same way.

3. there must be a column named "pattern" which is a SMARTS string describing an atom type.

The most important member functions of `typing\_rule` is the constructor and function `get`:

```
typing_rule( const char* filename );

const char* get(const atom_i& atom,const char* column) const;
```

which for a given atom search on the lines in the typing rule file from back to front, and stops at the one whose pattern matches atom's environment, and return the specified column.

OK, let's go back to the introductions to Hou's method. In fact, it is much more complicated than a simple atomic additive approach,some correction factors have been introduced to handle the complexity of soluability. Firstly there is a correction based on molecular weight, which is 0.0000075*mw*mw, this is imported as the estimation of surface area which should be linear to mw square. Another correction they introduced is the penalty on hydrophobic atoms, here hydrophobic atoms are defined as atoms which has no polar neighbours up to 3 degree, (a polar atom is an atom which is neither carbon nor hydrogen, a n-degree neighbour is an atom whose shortest path to target is n atoms long). If the hydrophobic atom is in a ring, the penalty would be -0.128, otherwise it would be -0.255.

Here is the interface for ring detection in mort:

```
  bool has_ring(const pointer_t& ptr, int size);

  bool find_ring(const pointer_t& ptr, int size, atomvec_t& ring);

  bool find_arom(const pointer_t& ptr, atomvec_t& arom);
```

we can detect if there is a ring or a certain sized ring, or even an aromatic ring (via Huckel's 4n+2 rule).

The following function to count molecular weight:

```
double count_weight( const molecule_t& mol )
{
  double weight = 0.0;

  moiter_t ai = mol.atom_begin();
  for( ; ai != mol.atom_end(); ++ai )
  {
    weight += ai->get_i(WEIGHT);
  }

  return weight;
}
```

The following function `is\_hyphb` tests for a hydrophobic atom, it uses a breadth first visit (BFV) algorithm to visit neighbours in certain degree:

```
bool is_polar( const atom_i& atom )
{
  int element = atom.get< element_t >();
  return element != HYDROGEN && element != CARBON;
}

bool is_hyphb( const atom_i& root, int degree )
{
  atomvec_t nbrs;
  nbrs.push_back( root );

  int start = 0;
  for( int i=0; i <= degree; ++i )
  {
    int end = nbrs.size();

    for( int j=start; j < end; j++ )
    {
      if( is_polar( nbrs[j] ) )
      {
        return false;
      }

      if( i == degree )
      {
        continue;
      }

      atom_t next = nbr_begin( nbrs[j] );
      for( ; next != nbr_end( nbrs[j] ); ++next )
      {
        if( nbrs.find(next)==nbrs.end())
        {
          nbrs.push_back( next );
        }
      }
    }
  }

  return true;
}
```

and the following is the rest part of code of logS:

```
double get_logs( const molecule_t& mol,
                 const typing_rule& rule,
                 const map< string, double >& parm )
{
  double sum = 0.0;
  double corr_hyb = 0.0;

  atom_t atom = atom_begin( mol );
  for( ; atom != atom_end( mol ); ++atom )
  {
    string type = rule.get( atom, ``typeid'' );

    double con = parm[ type ];
    sum += con;

    if( is_hyphb( atom, 4 ) )
    {
      if( has_ring( atom ) )
      {
        corr_hyb += 0.5 * parm[ ``HYD'' ];
      }
      else
      {
        corr_hyb += parm[ ``HYD'' ];
      }
    }
  }

  double mw = count_weight( mol );
  double corr_mw2 = parm[ ``MW2'' ] * mw * mw;
  double logs = parm[ ``RES'' ] + sum + corr_hyb +
                corr_mw2;

  return logs;
}

int main( int argc, char** argv )
{
  if( argc != 3 )
  {
    std::cerr << ``usage: hlogs molecule log'';
    std::cerr << std::endl;
    return -1;
  }

  try
  {
    typing_rule rule( ``typing.txt'' );

    map< string, double > parm;
    load_parm( ``logS.prm'', parm );

    int format = get_fmt( argv[1] );

    ifstream is( argv[1] );
```

```
      if( ! is )
      {
        throw logic_error(``Error: file ''+
                          string( argv[1] )+
                          `` does not exist. '' );
      }

      ofstream log( argv[2] );

      if( ! os )
      {
        throw runtime_error(``Error: can not open file ''+
                          string(argv[2])+
                          `` for written. '' );
      }

      nmol = 0;
      while( is )
      {
        molecule_t mol;
        read_mol( is, mol, format );

        if( atom_number( mol ) > 0 )
        {
          nmol++;
          double logs = get_logs( mol, rule, parm, log );

          cout <<``molecule '';
          cout <<setw( 7  )<<nmol << `` '' ;
          cout <<setw( 60 )<<mol.get<name_t>()<< `` '';
          cout <<setw( 15 )<<logs << std::endl;
        }
      }

      std::cout << ``Calculation done.\n'';
      return 0;
    }
    catch( std::exception& e )
    {
      cerr << e.what() << std::endl;
      return -2;
    }
  }
```

The full code could be found under directory *gleap/example/mort/hlogs*, with parameters and test sets.

3.5 makeles

In this section, we will introduce how to write a program to generate LES prmtop file used by sander.LES. We will learn about mort's ambmask interface, and will need modify mort's source code.

LES (Local Enhanced Sampling) is an important feature of AMBER, it involves make multiple copy of a flexible region, and run MD simulations on it to sample more conformations in limited time. To use LES, one should use the executable sander.LES instead of sander, and should use a modified prmtop file, whose format is slightly different from regular prmtop file.

There is a program named addles coming with AMBER which can be used to generate LES prmtop file, but it has some limitations. Firstly, addles can only use a regular prmtop file as input which is inconvenient; Secondly, all arrays in addles are declared statically which limited its usage on big systems; Thirdly, addles is originally written in fortran and is hard to modify and extend.

Because of these problems, there is strong motivation to replace addles with some new program, in this section we will illustrate how to write a program will same functionality but simpler and more extendable using mort. A fully functional addles should be able to handle multiple copy region cases, but here we only implement single copy region for simplicity.

The hardest part of addles is make copies of a select region. Firstly, we need a method to select a region of a big system,while we chose to use mort's ambmask interface for its general usage in amber. The function we are going to use is `mask\_atom`,

```
atomvec_t mask_atom(const molecule_t& mol,const string& mask);
```

which takes a molecule and a mask string as arguments and return an `atomvec` containing the atoms fitted to the mask. Secondly,we need make copies on the select region and make notes on the copied atoms, two hash code `LESTYPE` and `LESCOPY` has been defined for future access to parameter "lestype" and "lescopy". Parameter "lescopy" is used to record the atom's copy index, while "lestype" is used in multiple copy cases which we are not going to implement in this illustration version. Here we listed the source code of function make copy, which is acturally doing the copy job:

```
  shared_ptr< molecule_t > make_copy(const molecule_t& src,
                                     const string& mask,
                                     int ncopy )
  {
    atomvec_t region = mask_atom( src, mask );
    for_each(src.atom_begin(), src.atom_end(),
      iparm_setter(LESTYPE,0) );
    for_each(region.begin(),   region.end(),
      iparm_setter(LESTYPE,1) );

    map< int, int > idmap;
    shared_ptr< molecule_t > dst( new molecule_t() );
    moiter_t src_atom = atom_begin( src );
    for( ; src_atom != atom_end( src ); ++src_atom )
    {
      if( find(region.begin(),region.end(),src_atom)==
               region.end() )
      {
        moref_t dst_atom = dst->create_atom();
        copy_allparms( src_atom, dst_atom );
        dst_atom.set_i(LESCOPY, 0);
        idmap[ src_atom.getoid() ] = dst_atom.getoid();
      }
      else
      {
        for( int i=0; i < ncopy; ++i )
        {
          moref_t dst_atom = dst->create_atom();
          copy_allparms( src_atom, dst_atom )
          dst_atom.set_i(LESCOPY, i+1);
          if( i == 0 )
          {
            idmap[ src_atom.getoid() ] = dst_atom.getoid();
          }
        }
      }
    }
    moiter_t src_bond = src.bond_begin();
    for( ; src_bond != src.bond_end(); ++src_bond )
    {
      moref_t src_atm1 = atom_1st( *src_bond );
      moref_t src_atm2 = atom_2nd( *src_bond );

      bool copied1 = (find(region.begin(),region.end(),
                           src_atm1 )!=region.end());
      bool copied2 = (find(region.begin(),region.end(),
                           src_atm2 )!=region.end() );

      if( copied1 || copied2 )
      {
        int dst_id1 = idmap[ src_atm1.getoid() ];
        int dst_id2 = idmap[ src_atm2.getoid() ];

        for( int i=0; i < ncopy; ++i )
        {
          moref_t dst_atm1( *dst,ATOM, dst_id1 );
          moref_t dst_atm2( *dst,ATOM, dst_id2 );

          moref_t dst_bond = create_bond(dst_atm1,dst_atm2);
          copy_allparms( src_bond, dst_bond );

          if( copied1 ) dst_id1++;
          if( copied2 ) dst_id2++;
        }
      }
      else
      {
        int dst_id1 = idmap[ src_atm1.getoid() ];
        int dst_id2 = idmap[ src_atm2.getoid() ];

        moref_t dst_atm1( *dst, ATOM, dst_id1 );
        moref_t dst_atm2( *dst, ATOM, dst_id2 );

        moref_t dst_bond = create_bond(dst_atm1,dst_atm2);
        copy_allparms( src_bond, dst_bond );
      }
    }

    return dst;
  }
```

OK, it is understandable that you got a lot of questions about the above code, again, we are going to do the explanation line by line. Here comes the first three lines:

```
  atomvec_t region = mask_atom( src, mask );

  for_each( atom_begin( src ), atom_end( src ),
    iparm_setter(LESTYPE,0) );

  for_each( region.begin( ),   region.end( ),
    iparm_setter(LESTYPE,1) );
```

Line 1 is just calling `mask_atom` to get the selected region. Line 2 and line 3 worth more explanation. `iparm_setter` is a functor used for setting integer parameter of an object. it takes two argument to declare, the parameter ID and the parameter value.

Thus, now you should totally understand the code, which is acturally identical to the following:

```
iterator_t ai = mol.atom_begin();
for( ; ai != mol.atom_end(); ++ai )
{
  ai->set_i(LESTYPE, 0);
}

for( int i=0; i < region.end( mol ); ++i )
{
  region[i].set_i(LESTYPE, 1);
}
```

for the region going to be copied, the lestype should be 1, for the rest of the molecule, lestype should remain to be 0.

Here comes the part to copy atoms:

```
map< int, int > idmap;
shared_ptr< molecule_t > dst( new molecule_t() );
moiter_t src_ai = src.atom_begin();
for( ; src_ai != src.atom_end(); ++src_ai )
{
  if( region.find(*src_ai)==region.end() )
  {
    moref_t dst_atom = dst->create_atom();
    copy_allparm(*src_ai, dst_atom);
    dst_atom.set_i(LESCOPY, 0);
    idmap[ src_atom.absid() ] = dst_atom.absid();
  }
  else
  {
    for( int i=0; i < ncopy; ++i )
    {
      moref_t dst_atom = dst->create_atom();
      copy_allparms(*src_ai, dst_atom);
      dst_atom.set_i(LESCOPY, i+1);
      if( i == 0 )
      {
        idmap[src_atom.absid()]=dst_atom.absid();
      }
    }
  }
}
```

which is easy to understand, for each atom in the source molecule, if it is not in the copy region, make a new atom in the destinate molecule `dst` copy the parameters from source atom to destine atom (`dst_atom`), make a new entry in the ID map (`idmap`), let the source atom's absolute ID mapping to the destine atom's ID number; if it is in the copy region, make ncopy atoms in the destine molecule, copy parameter from source atom to every atom, let the abosolute ID of source atom mapping to the absolute ID of the first copied atom.

Then we can make copy of bonds:

```
iterator_t src_bi = src.bond_begin();
for( ; src_bi != src.bond_end(); ++src_bi )
{
  pointer_t src_atm1 = atom_1st( *bi );
  pointer_t src_atm2 = atom_2nd( *bi );

  bool copied1 = ( find(region.begin(),region.end(),
                     src_atm1 ) != region.end() );
    bool copied2 = ( find(region.begin(),region.end(),
                     src_atm2 ) != region.end() );

    if( copied1 || copied2 )
    {
      int dst_id1 = idmap[ src_atm1.getoid() ];
      int dst_id2 = idmap[ src_atm2.getoid() ];

      for( int i=0; i < ncopy; ++i )
      {
        pointer_t dst_atm1( *dst, ATOM, dst_id1 );
        pointer_t dst_atm2( *dst, ATOM, dst_id2 );

        pointer_t dst_bond=create_bond(dst_atm1,dst_atm2);
        copy_bond( src_bond, dst_bond );

        if( copied1 ) dst_id1++;
        if( copied2 ) dst_id2++;
      }
    }
    else
    {
      int dst_id1 = idmap[ src_atm1.getoid() ];
      int dst_id2 = idmap[ src_atm2.getoid() ];

      pointer_t dst_atm1( *dst, ATOM, dst_id1 );
      pointer_t dst_atm2( *dst, ATOM, dst_id2 );

      pointer_t dst_bond=create_bond(dst_atm1,dst_atm2);
      copy_bond(src_bond,dst_bond);
    }
  }
}
```

For each bond in source molecule (src), first get the two ending atoms of it (`src_atm1` and `src_atm2`), see if they are copied (`copied1` and `copied2`), look for the coresponding destine atom's absolute ID in the ID map (`dst\_id1` and `dst\_id2`). If the two atoms are both not copied, copy the bond only one time, otherwise copy the source bond `ncopy` times, increase destine atom's ID number by 1 each time if the corresponding source atom is copied.

OK, we have done the hardest part, but there are still several problems we need to figure out. First thing need to be fixed is the atom's nonbond exclusion list. As we know, there is an exclusion list in prmtop file, if we do not want calculate the nonbond interaction energy between two atoms, we just put the higher one's ID into lower one's exclusion list. Since the copied atoms doesn't know the existence of other copy, we surely want to put the ID of these atoms into exclusion list; Secondly, the angle list and torsion list also need to be fixed to avoid angles and torsions between different copy, and there force field parameter need to be scaled by ncopy. Thirdly we need prmtop file containing the les information, i.e. then lestype and lescopy parameter should be dumped into prmtop file.

Two achieve these goals, we need to modify mort's source code. As the first step, we explain here the procedure in mort to generate amber prmtop file, which is a little bit more complicated than usual file generate, the following functions are involved in this proceduce:

```
void load_amber_ffparam(istream& is,molecule_t& frc);

void exclude(const molecule_t& mol,excl_t& excl,
             int level);
```

```
void parametrize(molecule_t& mol,const
                 molecule_t& frc,
                 parmset_t& ps );

void write_amber_prmtop(ostream& os,
                        molecule_t& mol,
                        const parmset_t& ps );
```

first one load amber force filed parameter file (such as parm99.dat), save it in a molecule (not superisingly,since a force field is like a virtual molecule, each atom type is like an atom, has bonds, angles and torsions between them, thus a molecule would be a perfect place to hold these information).

Function exclude is making the exclusion list, in which we are interested. If you trace down it, you will found out it simply calls exclude_atom for each atom, and exclude_atom calls build_list to generate the exclusion list then sort it into a descenting order. The funciton build list is the one we want to modify, which located in file *gleap/mortsrc/prmtop/parm. cpp*, the following is the source code of build_list:

```
void build_list( const pointer_i& atom,
                 vector< int >& list,
                 vector< int >& dist,
                 vector< int >& visited,
                 int max_level )
{
  atomvec_t nbrs;
  nbrs.push_back( atom );

  int level = 1;
  int start = 0;
  while( level <= max_level )
  {
    int end = nbrs.size();

    for( int i=start; i < end; i++ )
    {
      iteratr_t nbr = nbrs[i].related_atom_begin(  );
      for( ; nbr != nbrs[i].related_atom_end( ); ++nbr )
      {
        if(find(nbrs.begin(),nbrs.end(),nbr)==
           nbrs.end() )
        {
          nbrs.push_back( nbr );

          if( nbr.getoid() > atom.getoid() )
          {
            list.push_back( nbr.get< id_t >() );
            dist.push_back( level );
          }
        }
      }
    }

    start = end;
    level++;
  }
}
```

as we may have noticed, this function is similar to function `is\_hyphb` we made in last section, which also used a BFS algorithm. what we want to do is add a function call `exclude\_les` at the end, which for copied atoms, put atoms from other copy into its exclusion list:

```
void exclude_les( const pointer_t& atom,
                  const vector< int >& list,
                  const vector< int >& dist )
{
  int lestype;
  if( atom.get_i(LESTYPE,lestype) && lestype>0 )
  {
    iterator_t excl = atom.getmol().atom_begin( );
    for( ; excl != atom.getmol().atom_end( ); ++excl )
    {
      if(excl.get_i(LESTYPE) !=atom.get_i(LESTYPE))
      {
        continue;
      }

      if( excl.get_i(LESCOPY)< atom.get_i(LESCOPY) )
      {
        continue;
      }

      if(find(list.begin(),list.end(),excl.get_i(ID))==
         list.end() )
      {
        list.push_back( excl.get_i(ID) );
        dist.push_back( 9 );
      }
    }
  }
}
```

Another function invovled is `parametrize`, which did a lot of things, first, it generate angles, and torsion and out-of-plane stretch (improper torsion), then find their parameters in the force field,store the parameters in a struct `parmset\_t`, and set each object a parameter “typeid” pointing to stored parameter.

Several things need to be fixed in the function, firstly in the stage of generating angles and torsions, we need to forbid the generation of angles and torsions between atoms from differnet copy. To assist this, we define the following function les_forbid to see if two atoms are forbiden from seeing each other :

```
  bool les_forbid(const atom_i& atm1,const atom_i& atm2)
  {
    if( ! atm1.has_i(LESTYPE) )
    {
      return false;
    }

    if(atm1.get_i(LESTYPE)!=atm2.get_i(LESTYPE))
    {
      return false;
    }

    return atm1.get_i(LESCOPY)!=atm2.get_i(LESCOPY);
  }
```

The functions acturally generating angles and torsions are defined in file *gleap/mortsrc/amb fmt/extent.cpp*, they are:

```
  void atom2angl(const molecule_t& ff,const atom_i& cent );
```

```
void bond2tors(const molecule_t& ff,const bond_i& bond );

void atom2oops(const molecule_t& ff,const atom_i& cent );
```

which as can be implied by name, and find and generate angles from atom, and generate torsions from bond. Why don't we generate angles by extenting bonds? The answer is we want to avoid double counting. For example, the angle A-B-C can be both reach by extenting bond A-B and bond B-C, which will cause double counting. This will not happen if we chose to extent from an atom, since an angle has only one central atom. The similar thing happens to proper and improper torsions, they are reached by extent bond and atom, respectively. Here we list part of the revised function atom2angl, to help you understand

```
void atom2angl(const molecule_t& ff,const atom_i& cent)
{
  iterator_t atm1 = cent.related_atom_begin();
  iterator_t aend = cent.related_atom_end();
  for( ; atm1 != aend; ++atm1 )
  {
    iterator_t atm2 = atm1;
    for( ++atm2; atm2 != aend; ++atm2 )
    {
      if( les_forbid( *atm1, *atm2 ) )
      {
        continue;
      }

      parm_angl( ff, atomvec_t( *atm1, cent, *atm2 ) );
    }
  }
}
```

The part with `les_forbid` is newly added, which skips th angles whose two ending atom are from different copy. Similar revisions has been made for function `bond2tors` and `atom2oops`.

Another thing need to be handled is the force filed parameters, for the copied bond, angle and torsions their energy need to be scaled by ncopy, so does the atom's VDW parameter and partial charge. This will involve some change of object's typeid and the parameters in parmset. First of all, we need distinguish objects whose parameters need to be fixed, which means go through the atoms of the object looking for atom with a non-zero lestype, if found, the object is copied and its parameter need to be fixed,

```
void set_les( pointer_t& obj )
{
  iterator_t atom = obj.related_atom_begin();
  for( ; atom != obj.related_atom_end(); ++atom )
  {
    int lestype;
    if( atom.get_i(LESTYPE,lestype) && lestype > 0 )
    {
      obj.set<lestype_t>(lestype);
    }
  }
}
```

Thus we have the following function `les_scale_each` defined to do the scale job,

```
void les_scale_each( pointer_t& obj, int ncopy  )
{
  double force = obj.get_d(FORCE);
  obj.set_d(FORCE, force/ncopy);
}
```

which for each copied object, looking for its typeid in idmap, if not found then and a new entry in the parms with scaled parameter, then set new typeid. Call this funtion with each object then we are done with the parameter scaling job.

Last thing we need to do is dumping the lestype and lescopy information , which invovles writing a new function `write_les` and call it from `write_amber_prmtop` in file *gleap/mortsrc/format/prmtop.cpp.* and we also need modify function `write_head`, since there is a tag in the head section of prmtop file indicating if this prmtop is a les prmtop. These code are all routine code, so we are not going to show the code here, you can find that in mortsrc if you are interested.

Chapter 4

Implementing Details of MORT

4.1 General introduction

Before demostate the examples, we would like to give some general introduction to some most frequently used mort classes and functions. As should be noted here, all these classes and functions are defined inside namespace "mort", thus you need include the following statement "`using namespace mort;`"in you program before you can use mort.

4.1.1 Naming conventions

In mort, all names are in lower case, words are connected using _, usual suffix includes `_t` denotes a type, `_i` denotes an interface which is a type with virtual functions and need to be inherited, `_T` denotes a template class.

4.1.2 Code orgnization

The source code of mort is orgnized as a collection of modules, each module include a sub-directory and a header file, currently there are following modules in mort:

common/ and *common.hpp*	This module contains all the basic operations, such as contant defintion, string algorithms and linear algebra algorithms.
object/ and *object.hpp*	This is the most important module, it contains all the basic types of mort, like molecule_t, moref_t, moiter_t, etc.
format/ and *format.hpp*	This module contains all kinds of molecular file format IO function.
smarts/ and *smarts.hpp*	This module contains an implementation of SMARTS language interpreter
atmask/ and *atmaks.hpp*	"atmask" is the short of "atom mask", it contains an implementation of amber mask interpreter.
tripos/ and *tripos.hpp*	Tripos's mol2 format interpreting subroutines, used by subroutines defined in module *format*.
pdbent/ and *pdbent.hpp*	PDB format interpreting functions.
ambfmt/ and *ambfmt.hpp*	Amber's file format interpreting functions, includes OFF lib format, force field, paramter format, prmtop format.

4.2 Hash code

If you take a look at the file *mortsrc/common/hashcode.hpp*, you will found a lot of integer constant definitions and two functions `hashid_t hash(const string&)` and `string unhash(const hashid_t&)`, like the following:

```
    typedef long long hashid_t;
    using std::string;
    static const hashid_t AAPOS            = 30905712LL;
    static const hashid_t ADDITION          = 1049725105308LL;
    ............
    static const hashid_t WGTDIFF           = 11199844318LL;

    string unhash(hashid_t id);
    hashid_t hash(const string& name);
```

The function `hash()` converts a string into a 64 bits integer, and function `unhash()` does the reverse operation, Here type `hashid_t` is a 64 bits integer type, it is defined as `long long`, as we has tested it works on both 32 bits linux and 64 bits linux, but you may have to change it to `int64` or something else for your native operation system. It is obvious that a 64 bits integer can not cover a string of any length, and yes we do have some limitations on input string. Firstly it can not be longer than 12 characters, secondly only lower cased characters and digits are allowed in this string.

The hash code generated using `hash()` are in most cases served as identities of parameters and components. Parameter and component are two important concepts of mort, and will be explained in detail in the coming sections, right now you just need know that a molecule could have multiple components, and an object (atom, bond, residue, etc.) could have multiple parameters, and retriving a component or a parameter by its identity is a very common operation in mort, thus to use an integer as identity will be much more efficient than using a string.

Now you might understand that both a string and an integer can be used to identify a parameter or a component. From now on, we will call the string as the parameter's or component's name, and call the integer as the parameter's or component's ID.

There are some very commonly used components (for example, atom, bond) and parameters (for example element, position). They are used so frequently in MORT that we decided to pre-calculate their ID and store them in a place to save the energy of future calculation. These are the constants you have seen in file *mortsrc/hashcode.hpp*. For example, the constant `ATOM` equals to `hash("atom")`.

It is very important for you to know that file *hashcode.hpp* is not hand written but generated using *hashgen.py* in the same directory. It takes the file *hash.txt* (also in the same directory) as input, and generated a C++ header file,and a python file which is for the python binding of MORT we are working on. The usage is like:

```
     hashgen.py -i hash.txt -op hashcode
```

In the command, hash.txt is the input which is a text with one string per line, hashcode is the output prefix.

The following code is taken from the test cases of hash, and it may help you better understand the usage and behavior of *hash()* and *unhash()*:

```

    BOOST_CHECK_EQUAL( ATOM, mort::hash("atom");
    BOOST_CHECK_EQUAL( unhash(ATOM), "atom" );
```

Some notes about mort's hash mechanism:

It is argubly that enumeration types can be used to archieve the same effect. We do agree, but enumeration types do have its advantage. As is well known, enumeration types are acturally integer types with accesending order of value. Thus, there is a chance to mess up them with normal integers, and causes an almost untracable bug. besides, user will need to update *hash.txt* and *hashcode.hpp* if they want to add an ID, which will cause the recompilation of the whole library, since allmost every source code file directly or indirectly included *hashcode.hpp*. With this hash mechanism, use can define their own ID set which will not collide with others, since same string always give same ID.

4.3 numeric vector types and eometrical algorithms

Class `numvec` as the name suggested, is "numeric vector" type of MORT, it is defined in file *mortsrc/common/numvec.hpp* and has been widely used in mort for the representation of positions, normals, etc. It inherits from type `boost::numeric::ublas::vector<double>`, and we have implemented some geometrical algorithms as listed below,

```
  void normalize( numvec& vec );
```

```
numvec normalize_copy( const numvec& v );
void rotate(const numvec& vec, const numvec axis,
            double theta);
numvec rotate_copy(const numvec& vec, numvec axs,
                   double theta);
double dotpod(const numvec& v1, const numvec& v2 );
numvec cross( const numvec& v1, const numvec& v2 );
double norm( const numvec& vec );
double dist( const numvec& v1, const numvec& v2 );
double dis2( const numvec& v1, const numvec& v2 );
double angl( const numvec& v1, const numvec& v2, const numvec& v3);
double tors( const numvec& v1, const numvec& v2,
             const numvec&v3, const numvec& v4);
double max(const numvec& v1);
double min(const numvec& v2);
```

These routines are all self-explained. The following code taken from test cases of `numvec` may help user better understand of the usage of `numvec`:

```
numvec p1(3);
p1[0] = 1.0;
p1[1] = 2.0;
P1[2] = 3.0;

numvec p2(0.0, 1.0, 2.0);
p1 -= p2;
BOOST_CHECK_EQUAL( p1[0], 1.0, 1e-6);
BOOST_CHECK_EQUAL( p1[1], 1.0, 1e-6);
BOOST_CHECK_EQUAL( p1[2], 1.0, 1e-6);

numvec p3 = p1 + p2*2.0;
BOOST_CHECK_EQUAL( p3[0], 1.0, 1e-6);
BOOST_CHECK_EQUAL( p3[1], 4.0, 1e-6);
BOOST_CHECK_EQUAL( p3[2], 7.0, 1e-6);

BOOST_CHECK_EQUAL( dist(p1,p3), 2.0*normal(p2), 1e-6);
```

4.4 Parameter and entity

Parameters as mentioned before, is a very important concept of MORT, and is easy to understand too. Basically, it is just a variable with an identity(ID or name) and can be set to an entity or an object (entity can be considered as a container for a lot of parameters, object is a little more complicated and will be explained later) and latter be retrived by its ID (or name). Entity (class `entity_t`)as has been mentioned can be considered as a parameter container, which contains and ID-¿value one to one map. Here we listed the member functions of `entity_t`, and divide them into several group.

4.4.1 Common member function

This group contains the common member functions of `entity_t`, the constructor, the copy constructor, the assignment operator, etc.

```
entity_t();  /// \brief constructor

entity_t(const entity_t& rhs); /// \brief copy constructor
```

```
    entity_t& operator=( const entity_t& rhs ); /// \brief assignment opertaor

    virtual ~entity_t();    /// \brief deconstructor

    /// \brief swap the content of two entity
    ///
    /// very useful for exceptional safety.
    virtual void swap( entity_t& rhs );
```

Function `swap()` worths more discussion, it as the name indicated will swap the content of two entities. It is important because you can easily write exceptional safe code using it. An exceptional safe function will mantain the integrity of its input. If some unexpected error happened inside the function, it will not change the content of its input. You can easily archieve this by using `swap()`, generate a temporary entity, work on this temporary entity, so if something nasty happens, you input will not be effected, at the end of your function, swap the content of temporary entity and your input, this process is only change some pointer, so is fast and will not throw exception.

4.4.2 Setter functions

This group contains setter functions, which set parameter of entity.

```
    // series of functions to set paramters
    void set_i(const hashid_t& parmid, int value);
    void set_d(const hashid_t& parmid, double value);
    void set_s(const hashid_t& parmid, const string& value);
    void set_v(const hashid_t& parmid, const numvec& value);
    void set_a(const hashid_t& parmid, const any& value);

    void set_i(const string& parmname, int value);
    void set_d(const string& parmname, double value);
    void set_s(const string& parmname, const string& value);
    void set_v(const string& parmname, const numvec& value);
    void set_a(const string& parmname, const any& value);
```

As it can be seen, paramters are classified into five categories: integer paramters, double parameters, string parameter, numvec parameters and parameters of any other type are in the last category. The storage strategy for the first four types of parameters are simple and easy to understand, we use a `hash_map` to save a one to one map of parameters. As for the last category, since we have already explained the mechanism of `boost::any`, it is not hard to figure it out, parameters of any type will be firstly converted into `boost::any`, stored inside `entity_t` in a `hash_map<int,boost::any>` structure, and user can later retrive it back and converted it to the original type using `boost::any_cast`. One might will will question why we are not using a single template function here to replace all these functions to avoid complexity? Acturally, that is what we did in the earlier version of MORT, later we realized a very long compilation time, since the seperative compilation of template has not been supported by major C++ compilers, so we have to include all the implementation in header file, which is a burden since `entity_t` is inherited by a lot of types. Another argument here is why we are not using `boost::any` to intermediate every type, this is mainly an efficiency issue. According to our experience, about 95% of paramters are of the first four data type, thus, we could get best efficiency be specialize the operation of these four types.

Besides, you can set parameter by its name or ID.

4.4.3 Type I getter functions

This group contains so called "type I getter function", which takes two arguments: the identity of parameter (ID or name) and a reference to output (pointer or value, the latter one is mort convienent but less efficient), returns a bool variable reports if the return value is true, means the parameter has been defined and pointer has been modified to be pointing to the parameter, otherwise the parameter has not been defined and the pointer will not be changed.

```
    // series functions to get parameters
    bool get_i( int parmid, int& value ) const;
    bool get_d( int parmid, double& value ) const;
    bool get_s( int parmid, string& value ) const;
    bool get_v( int parmid, numvec& value ) const;
    bool get_a( int parmid, any& value ) const;

    bool get_i( const string& parmname, int& value ) const;
    bool get_d( const string& parmname, double& value ) const;
    bool get_s( const string& parmname, string& value ) const;
    bool get_v( const string& parmname, numvec& value ) const;
    bool get_a( const string& parmname, any& value ) const;

    bool get_iptr( const hashid_t& parmid, const int*& value ) const;
    bool get_dptr( const hashid_t& parmid, const double*& value ) const;
    bool get_sptr( const hashid_t& parmid, const string*& value ) const;
    bool get_vptr( const hashid_t& parmid, const numvec*& value ) const;
    bool get_aptr( const hashid_t& parmid, const any*& value ) const;

    bool get_iptr( const hashid_t& parmid, int*& value );
    bool get_dptr( const hashid_t& parmid, double*& value );
    bool get_sptr( const hashid_t& parmid, string*& value );
    bool get_vptr( const hashid_t& parmid, numvec*& value );
    bool get_aptr( const hashid_t& parmid, any*& value );
```

This type of functions usually used when your are not sure if some parameter is defined or not.

4.4.4 Type II getter function

This type of getter functions takes parameter identity and return reference of the parameter value. if the parameter has not been defined, it will throw out an exception.

```
    int& get_i( const hashid_t& parmid );
    double& get_d( const hashid_t& parmid );
    string& get_s( const hashid_t& parmid );
    numvec& get_v( const hashid_t& parmid );
    any& get_a( const hashid_t& parmid );

    const int& get_i( const hashid_t& parmid ) const;
    const double& get_d( const hashid_t& parmid ) const;
    const string& get_s( const hashid_t& parmid ) const;
    const numvec& get_v( const hashid_t& parmid ) const;
    const any& get_a( const hashid_t& parmid ) const;

    int& get_i(const string& parmname);
    double& get_d(const string& parmname);
    string& get_s(const string& parmname);
    numvec& get_v(const string& parmname);
    any& get_a(const string& parmname);

    const int& get_i(const string& parmname) const;
    const double& get_d(const string& parmname) const;
    const string& get_s(const string& parmname) const;
    const numvec& get_v(const string& parmname) const;
    const any& get_a(const string& parmname) const;
```

This type of functions is usually used when you are sure a parameter has been defined.

4.4.5 Type III getter functions

This type of getter function is so called "force getter", mean it will define one if it cannot find the parameter, and then return the reference, use it only when you are sure about what you are doing.

```
    int& frcget_i(const hashid_t& parmid);
    double& frcget_d(const hashid_t& parmid);
    string& frcget_s(const hashid_t& parmid);
    numvec& frcget_v(const hashid_t& parmid);
    any& frcget_a(const hashid_t& parmid);

    int& frcget_i(const string& parmname);
    double& frcget_d(const string& parmname);
    string& frcget_s(const string& parmname);
    numvec& frcget_v(const string& parmname);
    any& frcget_a(const string& parmname);
```

4.4.6 Testing function

Testing functions test if certain parameter has been defined.

```
    bool has_i( const hashid_t& parmid ) const;
    bool has_d( const hashid_t& parmid ) const;
    bool has_s( const hashid_t& parmid ) const;
    bool has_v( const hashid_t& parmid ) const;
    bool has_a( const hashid_t& parmid ) const;
    bool has_i( const string& parmid ) const;
    bool has_d( const string& parmid ) const;
    bool has_s( const string& parmid ) const;
    bool has_v( const string& parmid ) const;
    bool has_a( const string& parmid ) const;
```

4.4.7 Iterator function

Iterator function return the iterator of internal parameter container.

```
    hash_map<int, int>::const_iterator ibegin() const;
    hash_map<int, int>::const_iterator iend() const;

    hash_map<int, double>::const_iterator dbegin() const;
    hash_map<int, double>::const_iterator dend() const;

    hash_map<int, string>::const_iterator sbegin() const;
    hash_map<int, string>::const_iterator send() const;

    hash_map<int, numvec>::const_iterator vbegin() const;
    hash_map<int, numvec>::const_iterator vend() const;

    hash_map<int, any>::const_iterator abegin() const;
    hash_map<int, any>::const_iterator aend() const;
```

4.4.8 Examples of entity usage

The following examples shows the usage of `entity_t`,

```
    entity_t e;
    e.set_i(ELEMENT, 7);
    BOOST_CHECK_EQUAL( e.get_i(ELEMENT), 7 );

    e.set_s("name", "C0");
    BOOST_CHECK_EQUAL( e.get_s(NAME), "C0");

    std::pair<int,int> ip1 = std::make_pair(3, 4);
    e.set_a(ATOMPAIR, ip1);
    std::pair<int,int> ip2 = boost::any_cast< std::pair<int,int> >( &e.get_a(ATOMPAIR) );
    BOOST_CHECK_EQUAL(ip2.first, 3);
    BOOST_CHECK_EQUAL(ip2.second,4);

    int* pelement=NULL;
    BOOST_CHECK( e.get_i(ELEMENT, pelement) );
    BOOST_CHECK_EQUAL(*pelement, 7);
    *pelement = 8;
    BOOST_CHECK_EQUAL(e.get_i(ELEMENT), 8);


    string* ptype=NULL;
    BOOST_CHECK_EQUAL( !e.get_s(TYPE, ptype) );

    try
    {
        e.get_s("type") = "CA";
        BOOST_CHECK( false );
    }
    catch( std::exception&  )
    {
        BOOST_CHECK( true );
    }

    e.frcget_s("type") = "CA";
    BOOST_CHECK_EQUAL( e.get_s("type"), "CA" );
    e.get_s("type") = "O";
    BOOST_CHECK_EQUAL( e.get_s("type"), "O" );
```

4.5 Molecule, Component, and Adjacency

These three classes molecule_t is the most important classed in MORT. Before introduce there member functions in detail, we would like to introduce the basic idea roughly. In mort, one molecule could have several components and several adjacencies. One component has several objects, and one object has been assigned an absoulte ID to desitinguish its position in the component, and has several parameters defined. As for adjacency, it is defined for a pair of components and records the relations between the objects in the two components.

Now you may be confused by this whole bunch of stuff. An example may help you better understand it. The following code illustrates how to constitue a mechane using MORT:

```
    molecule_t mol;
    component_t* atoms = mol.create_component(ATOM,5);
    component_t* bonds = mol.create_component(BOND,4);

    adjacency_t* a_a = mol.create_adjacency(ATOM,ATOM);
    adjacency_t* a_b = mol.create_adjacency(ATOM,BOND);
    adjacency_t* b_a = mol.create_adjacency(BOND,ATOM);
    atoms->set_s(NAME, 0, "C1");
    atoms->set_s(NAME, 1, "H2");
    atoms->set_s(NAME, 2, "H3");
    atoms->set_s(NAME, 3, "H4");

    atoms->set_i(ELEMENT, 0, 6);
    atoms->set_i(ELEMENT, 1, 1);
    atoms->set_i(ELEMENT, 2, 1);
    atoms->set_i(ELEMENT, 3, 1);
    atoms->set_i(ELEMENT, 4, 1);

    bonds->set_i(ORDER, 0, 1);
    bonds->set_i(ORDER, 1, 1);
    bonds->set_i(ORDER, 2, 1);
    bonds->set_i(ORDER, 3, 1);

    a_a->connect(0, 1); // connect atom0 to atom1
    a_a->connect(1, 0); // connect atom1 to atom0
    a_b->connect(0, 0); // connect atom0 to bond0
    b_a->connect(0, 0); // connect bond0 to atom0
    a_b->connect(1, 0); // connect atom1 to bond0
    b_a->connect(0, 1); // connect bond0 to atom1

    a_a->connect(0, 2); // connect atom0 to atom2
    a_a->connect(2, 0); // connect atom2 to atom0
    a_b->connect(0, 1); // connect atom0 to bond1
    b_a->connect(1, 0); // connect bond1 to atom0
    a_b->connect(2, 1); // connect atom2 to bond1
    b_a->connect(1, 2); // connect bond1 to atom2

    a_a->connect(0, 3); // connect atom0 to atom3
    a_a->connect(3, 0); // connect atom3 to atom0
    a_b->connect(0, 2); // connect atom0 to bond2
    b_a->connect(2, 0); // connect bond2 to atom0
    a_b->connect(3, 2); // connect atom3 to bond2
    b_a->connect(2, 3); // connect bond2 to atom3

    a_a->connect(0, 4); // connect atom0 to atom4
    a_a->connect(4, 0); // connect atom4 to atom0
    a_b->connect(0, 3); // connect atom0 to bond3
    b_a->connect(3, 0); // connect bond3 to atom0
    a_b->connect(4, 3); // connect atom4 to bond3
    b_a->connect(3, 4); // connect bond3 to atom4
```

With the above code, we constructed a methane. Firstly, two compnents has been created with the names `atoms` and `bonds`, the member function `create_component()` of `molecule_t` takes two arguments, the ID of component (as we discussed before) and the size (number of objects in it). As can be seen component `atoms` has 5 objects (or atoms) with their absolute ID to be 0 to 4, and component `bonds` has 4 objects(or bonds). with their absoluted ID to be 0 to 3 (the absolute ID assigning strategy will be discussed later).

Secondly, two adjacencis were created, one is between atom and atom with the name `a_a`, the other is between atom and bond with the name `a_b`. The member function `create_adjacency` of `molecule_t` takes two component ID as arguments and create an adjanecy between the two components.

Thirdly, objects' parameter is assigned according to its absolute ID. For each atom, two parameters are set: its name and its element (or atomic number). For each bond, only one parameter is set: its order.

At the last step, the connection between objects are generated by using the member function `connect()` of `adjacency_t`, which takes the absolute ID of two objects as arguments, and make a connection between them.

In the code comments you can find details about the connection it has made.

4.5.1 Molecule

The following list is the member function list of `molecule_t`. Also noted here that molecule inherited from `entity_t`, so you can use all entity's member function for a molecule.

```
molecule_t();/// \brief constructor
molecule_t(const molecule_t& rhs);/// \brief copy constructor
molecule_t& operator=(const molecule_t& rhs);/// \brief assignment operator
virtual ~molecule_t();/// \brief deconstructor
virtual void swap( molecule_t& rhs );/// \brief swap the content of two molecule

/// \brief get the component with given ID
///
/// return blank component if such a component does not exist.
component_t const* get_component(int cid) const;

component_t* create_component(int cid, int size=0) const;

/// \brief force get component, add new one if necessary
/// the ID number should be generated by hash_T to avoid
/// conflict.
component_t* frcget_component(int cid);


/// \brief get adjacency, create new one if necessary
/// \param cid1 the first component
/// \param cid2 the second component
adjacency_t* frcget_adjacency( int cid1, int cid2 );

/// \brief get the adjacency for relations between components
/// \param cid1 the first component
/// \param cid2 the second component
adjacency_t const* get_adjacency( int cid1, int cid2 ) const;

///  get the list of connected components
bool get_nbrlist(int cid, const set<int>*& p) const;

int natom() const;
int nbond() const;
int nresd() const;
int nangl() const;
int ntors() const;
int ntor2() const;
int noops() const;
int nptor() const;

range_t atoms() const;
range_t bonds() const;
range_t resds() const;
range_t angls() const;
range_t torss() const;
range_t tor2s() const;
range_t oopss() const;
range_t ptors() const;

iterator_t atom_begin() const;
```

```
    iterator_t atom_end() const;
    iterator_t bond_begin() const;
    iterator_t bond_end() const;
    iterator_t resd_begin() const;
    iterator_t resd_end() const;
    iterator_t angl_begin() const;
    iterator_t angl_end() const;
    iterator_t oops_begin() const;
    iterator_t oops_end() const;
    iterator_t tors_begin() const;
    iterator_t tors_end() const;
    iterator_t tor2_begin() const;
    iterator_t tor2_end() const;
    iterator_t ptor_begin() const;
    iterator_t ptor_end() const;

    pointer_t create_atom();
    pointer_t create_bond();
    pointer_t create_resd();
    pointer_t create_angl();
    pointer_t create_tors();
    pointer_t create_tor2();
    pointer_t create_ptor();
    pointer_t create_oops();

    pointer_t frcget_atom(const string& name);
    pointer_t frcget_bond(const string& name1, const string& name2);
    pointer_t frcget_angl(const string& name1, const string& name2, const string& name3);
    pointer_t frcget_tors(const vector<string>& names, int period);
    pointer_t frcget_oops(const vector<string>& names, int period);
    pointer_t frcget_resd(const string& name);

    pointer_t get_atom(const string& name) const;
    pointer_t get_resd(const string& name) const;

    bool has_atom(const string& name) const;
    bool has_resd(const string& name) const;

    bool get_atom(const string& name, pointer_t& atom) const;
    bool get_resd(const string& name, pointer_t& resd) const;

    bool has_atom(ptrpred_t pred) const;
    bool has_bond(ptrpred_t pred) const;
    bool has_resd(ptrpred_t pred) const;
    bool has_angl(ptrpred_t pred) const;
    bool has_tors(ptrpred_t pred) const;
    bool has_tor2(ptrpred_t pred) const;
    bool has_ptor(ptrpred_t pred) const;
    bool has_oops(ptrpred_t pred) const;

    bool get_atom(ptrpred_t pred, pointer_t& atom) const;
    bool get_bond(ptrpred_t pred, pointer_t& bond) const;
    bool get_resd(ptrpred_t pred, pointer_t& resd) const;
    bool get_angl(ptrpred_t pred, pointer_t& angl) const;
    bool get_tors(ptrpred_t pred, pointer_t& tors) const;
    bool get_tor2(ptrpred_t pred, pointer_t& tor2) const;
    bool get_ptor(ptrpred_t pred, pointer_t& ptor) const;
    bool get_oops(ptrpred_t pred, pointer_t& oops) const;
    pointer_t get_atom(ptrpred_t pred) const;
    pointer_t get_bond(ptrpred_t pred) const;
    pointer_t get_resd(ptrpred_t pred) const;
    pointer_t get_angl(ptrpred_t pred) const;
    pointer_t get_tors(ptrpred_t pred) const;
    pointer_t get_tor2(ptrpred_t pred) const;
    pointer_t get_ptor(ptrpred_t pred) const;
    pointer_t get_oops(ptrpred_t pred) const;
```

Among these member functions, `create_component` and `create_adjacency` has been used and explained, `get_component` is used to retrive a component by its ID, in the component does not exist, it will return the pointer to a empty component. `frcget_component` is also used to retrive a component, the difference is, if the component does not exist, it will create such a one and return the pointer to it. Function `get_adjacency()` and `frcget_adjacency()` have similar meanings.

Theoritcally you can create any number of components and adjacencies within a molecule. In pratise, there are several components (seven acturally) which are most frequently used and `molecule_t` has include special member functions to ease the access of these components. The IDs and meanings of there components are: ATOM (for atom), BOND (for bond), ANGL (for angle), TORS (for torsion), OOPS (for Out-Of-Plan Stretch, or improper torsion), TOR2 (for torsion-torsion interaction), PTOR (PI-torsion). The last two is used by amoeba force field operation. For example, what member function `natom()` does is just `get_component(ATOM)->size()`. Right now you can just disregard those member functions relevant to type `pointer_t`, `iterator_t` and `range_t`, they will be covered in the next subsection.

4.5.2 Component

Before we going to the implementation details of class `component_t`, we would like to introduce its principal. As we have mentioned before, one component have multiple objects, each of these objects when created was assigned an absolute ID. These absoulte IDs are stored in the data member of `component_t` named `m_absids` of type `vector<int>`, and assigned in an "always ascending" order, i.e. there is an ID counter in a component, which will increase by one every time a new object is created, but will not decrease if an object is deleted, and object's absolute ID is assigned based on this counter `component_t` has two member functions `absid_begin()` and `absis_end()` for the iterations of absolute IDs.

An object has multiple parameters, however in the implementation,the parameters of same of object is not stored together, instead, to archieve the best efficiency and memory layout, parameters of different object but of same identity are stored together in a linear table. Take methane as an example, you might still remember in the example code we set the element (atomic number) of each atom, in reality they are store like the following. The atomic component (named `atoms` in our example) has a data member `m_iparms`, which is a one to one, parameter ID to linear table map. Inside it, there is an entry: parameter ID ELEMENT is pointing to a linear table (of type `vector<int>`) which has the value `{6,1,1,1,1}`.

Now you have some basic idea of how `component_t` works, but it is not the whole story. As you may have noticed, the term "absolute ID" indicates there may exists some kind of "relative ID". Yes, in the dictionary of MORT, object's "relative ID" is the index of its absolute ID in the component's absolute ID array (remember the data member `m_absids`?) These two IDs is different only if some object has been deleted. It is because when we are deleting an object from a component, we are not going to each parameter array, remove its parameter from the parameter array, instead we just simply remove its absolute ID from absolute ID array. Only if object deletion happens in a component, an object's absolute ID could be different from its relative ID. For example, if we delete the last atom (atom 4) of mechane, and then create a new one, the new atom's ID will be 5 not 4, even if atom 4 has been deleted. It is very important for you to keep in mind that parameter is stored according to absolute ID. For example, if atom 4 is deleted from methane, the parameter array of ELEMENT will remain the same, i.e. still be `{6,1,1,1,1}`, just the last element will never be referenced anymort, i.e. it has be deserted (kind like the situation of "memory leak")

There are a lot of advantages to implement object deletion in this way, firstly it will be a huge burden if we go into each parameter array to remove one item every we are trying to delete an object. secondly, even if we can do that there are a lot of work need to be done to clean up the adjacency, since all the object after the deleted one their ID has been changed.

But there is also some disadvantages. For efficiency reason, sometimes user wants to get the pointer of the parameter array, so they can perform some operation directly on the memory. If the component is “clean” (i.e. object’s relative ID equals absolute id), one can do this by calling member function `get_xptr()` (here x could be ’i’, ’d’, ’s’, ’v’, ’a’). One can not do that, if a component is not clean, since in that case, parameters is not in the same order of objects. Member function `cleanup()` is provided to fix this problem. It will rebuild the absolute ID array and all parameter value array to make the component clean again, and returns an old ID to new ID map for user to correct its relevant adjacency. It would be better that one use member function `islcean()` to test if a component is clean before calling this function.

As for the storage of parameter in component, it is almost the same as entity, we use a one to one parameter ID to parameter array map for it. It is slightly different for `numvec` typed parameter, instead of being stored in a `vector<numvec>` container, it is stored in a `vector<double>` container, also for efficiency reason.

Below is the list of member functions of `component_t`,

```
    /// \brief contructor
    /// \param size the number of objects in the component;
    component_t( int size = 0 );

    /// \brief copy constructor
    component_t( const component_t& rhs );

    /// \brief deconstructor
    ~component_t();

    /// \brief swap the content of two component
    void swap( component_t& rhs );

    /// \brief add new object to the end.
    iterator append( int n );

    /// \brief insert new object at the certain position.
    /// \param i the position where new object will be added.
    iterator insert( const iterator& pos, int n );

    /// \brief remove object with given ID number
    /// \param the ID number of the object going to be removed.
    bool remove( int id );

    /// \brief adjust the size of the component
    /// \param size the new size
    void resize( int size );

    /// \brief return the size of the component
    int size() const;

    iterator absid_begin() const;
    iterator absid_end() const;

    bool isclean() const;
    void cleanup(map<int, int>& old2new);

    // series of member fucntion set paramters of different type
    void set_i( const hashid_t& parmid, int absid, int value );
    void set_d( const hashid_t& parmid, int absid, double value );
    void set_s( const hashid_t& parmid, int absid, const string& value );
    void set_v( const hashid_t& parmid, int absid, const numvec& value );
    void set_a( const hashid_t& parmid, int absid, const any& value );
```

```
// series of member fucntion get paramters of different type
int& get_i( const hashid_t& parmid, int absid );
double& get_d( const hashid_t& parmid, int absid );
string& get_s( const hashid_t& parmid, int absid );
any& get_a( const hashid_t& parmid, int absid );

const int& get_i( const hashid_t& parmid, int absid ) const;
const double& get_d( const hashid_t& parmid, int absid ) const;
const string& get_s( const hashid_t& parmid, int absid ) const;
numvec get_v( const hashid_t& parmid, int absid ) const;
const any& get_a( const hashid_t& parmid, int absid ) const;

double* get_vptr( const hashid_t& parmid, int absid );
double const* get_vptr(const hashid_t& parmid, int absid) const;

bool get_i(const hashid_t& parmid, int absid, int& pi) const;
bool get_d(const hashid_t& parmid, int absid, double& pd) const;
bool get_s(const hashid_t& parmid, int absid, string& ps) const;
bool get_v(const hashid_t& parmid, int absid, double& pv) const;
bool get_a(const hashid_t& parmid, int absid, any& v) const;

bool get_iptr(const hashid_t& parmid, int absid, int*& pi);
bool get_dptr(const hashid_t& parmid, int absid, double*& pd);
bool get_sptr(const hashid_t& parmid, int absid, string*& ps);
bool get_vptr(const hashid_t& parmid, int absid, double*& pv);
bool get_aptr(const hashid_t& parmid, int absid, any*& pa);

bool get_iptr(const hashid_t& parmid, int absid, const int*& pi) const;
bool get_dptr(const hashid_t& parmid, int absid, const double*& pd) const;
bool get_sptr(const hashid_t& parmid, int absid, const string*& ps) const;
bool get_vptr(const hashid_t& parmid, int absid, const double*& pv) const;
bool get_aptr(const hashid_t& parmid, int absid, const any*& pa) const;


bool has_i( const hashid_t& parmid, int absid ) const;
bool has_d( const hashid_t& parmid, int absid ) const;
bool has_s( const hashid_t& parmid, int absid ) const;
bool has_v( const hashid_t& parmid, int absid ) const;
bool has_a( const hashid_t& parmid, int absid ) const;


int* getall_i( const hashid_t& parmid );
double* getall_d( const hashid_t& parmid );
string* getall_s( const hashid_t& parmid );
double* getall_v( const hashid_t& parmid );
any* getall_a( const hashid_t& parmid );

const int* getall_i( const hashid_t& parmid ) const;
const double* getall_d( const hashid_t& parmid ) const;
const string* getall_s( const hashid_t& parmid ) const;
const double* getall_v( const hashid_t& parmid ) const;
const any* getall_a( const hashid_t& parmid ) const;


iparm_iterator iparm_begin() const;
iparm_iterator iparm_end()   const;
dparm_iterator dparm_begin() const;
dparm_iterator dparm_end()   const;
sparm_iterator sparm_begin() const;
```

```
        sparm_iterator sparm_end()   const;
        vparm_iterator vparm_begin() const;
        vparm_iterator vparm_end()   const;
        aparm_iterator aparm_begin() const;
        aparm_iterator aparm_end()   const;
```

The getter functions of `component_t` is similar to those of `entity_t`, except three things. Firstly, an extra argument `absid` is required for objects absolute ID; secondly, no type III getter function, i.e. the "force getter"; thirdly, the operation of for `numvec` parameter is different, function `get_vptr` returns a `double` pointer instead of a `numvec` pointer, function `get_v` returns a `numvec` instead of a reference, and has no mutable version, these are all due to storage natural of `numvec`.

4.5.3 Adjacency

Class `adjacency_t` is much simpler compare to class `component_t`, the following is a full list of its member functions, all are straightfoward, nothing important need to emphasized here.

```
        adjacency_t( );

        /// \brief copy construction function
        adjacency_t( adjacency_t const& rhs );

        /// \brief deconstruction function
        ~adjacency_t( );

        /// \brief make connection between objects
        /// \param a  ID number of an object
        /// \param b  ID number of an object
        /// \return true if connection made successfully,
        ///         false if a and b are already connected.
        bool add( int a, int b );

        /// \brief test if connection exist between a and b
        /// \param a  ID number of an object
        /// \param b  ID number of an object
        /// \return test result if there is a connection between a and b.
        bool has( int a, int b ) const;

        /// \brief remove connection between a and b
        /// \param a  ID number of an object
        /// \param b  ID number of an object
        /// \return true if connection removed successfully,
        ///         false if there is no connection between a and b.
        bool remove( int a, int b );

        /// remove all connections of a
        void clear( int a );

        /// \brief get iterator pointing to the connection.
        /// \param a  ID number of an object
        /// \param b  ID number of an object
        /// \return the iterator pointing to the connection, usually this
        ///         function is used with begin() to give the ID the connection.
        /// \sa     begin()
        iterator find ( int a, int b ) const;

        /// \brief starting point of an object's connection
        /// \param id ID number of an object
        iterator begin( int id ) const;

        /// \brief ending point of an object's connection
        /// \param id ID number of an object
        iterator end  ( int id ) const;


        /// \brief number of an object's connection
        int size( int id ) const;
```

4.6 Pointer, iterator and range

Theoritcally, `molecule_t`, `component_t` and `adjacency_t` is enough for user to access the functionality of `molecule_t`, but in pratise, we found to use them directly is fairly inconvinent. Class `pointer_t`, `iterator_t` and `range_t` is provided to ease the procedure.

4.6.1 Pointer

Class `pointer_t` as the name suggested, is pointing to an object inside a molecule. To declare a pointer, you will need three arguments: the molecule, the component ID, and the object's absolute ID. There are some member functions of `molecule_t` returns a pointer, such as `create_atom` which create a new atom and returns pointer to it, and `get_atom(const string& name)` which search for an atom with given name inside the molecule, and returns a pointer to it. By using `pointer_t` the construction of methane can be rewriten as the following code:

```
    molecule_t mol;
    pointer_t atom0 = mol.create_atom();
    pointer_t atom1 = mol.create_atom();
    pointer_t atom2 = mol.create_atom();
    pointer_t atom3 = mol.create_atom();
    pointer_t atom4 = mol.create_atom();

    atom0.set_s(NAME, "C0");
    atom1.set_s(NAME, "H1");
    atom2.set_s(NAME, "H2");
    atom3.set_s(NAME, "H3");
    atom4.set_s(NAME, "h4");

    atom0.set_i(ELEMENT, 6);
    atom1.set_i(ELEMENT, 1);
    atom2.set_i(ELEMENT, 1);
    atom3.set_i(ELEMENT, 1);
    atom4.set_i(ELEMENT, 1);

    pointer_t bond0 = create_bond(atom0, atom1);
    pointer_t bond1 = create_bond(atom0, atom2);
    pointer_t bond2 = create_bond(atom0, atom3);
    pointer_t bond3 = create_bond(atom0, atom4);
```

As illustrated, `pointer_t` has its setter and getter functions which calls the setter and getter functions of `component_t` to set and get parameters, and it also has member function `connect()` which calls `adjacency_t`'s member function to make connection. In a sense, `pointer_t` is a wrapper class which packages all operations of an object together. The following is the full list of member functions of `pointer_t`:

```
  pointer_t( const molecule_t& mol, int compid, int objectid );
```

```
pointer_t( const pointer_t& rhs );

virtual ~pointer_t();

pointer_t& operator=( const pointer_t& rhs );

void absid(int id);
int absid(void) const;
int relid(void) const;
int cmpid(void) const;

molecule_t& getmol();
molecule_t const& getmol() const;
component_t* get_component();
component_t const* get_component() const;
adjacency_t* get_adjacency(int cid);
adjacency_t const* get_adjacency(int cid) const;

void set_i( const hashid_t& parmid, int value );
void set_d( const hashid_t& parmid, double value );
void set_s( const hashid_t& parmid, const string& value );
void set_v( const hashid_t& parmid, const numvec& value );
void set_a( const hashid_t& parmid, const any& value );

void set_i( const string& parmname, int value );
void set_d( const string& parmname, double value );
void set_s( const string& parmname, const string& value );
void set_v( const string& parmname, const numvec& value );
void set_a( const string& parmname, const any& value );

// type I getter function
bool get_i( const hashid_t& parmid, int& value ) const;
bool get_d( const hashid_t& parmid, double& value ) const;
bool get_s( const hashid_t& parmid, string& value ) const;
bool get_v( const hashid_t& parmid, numvec& value ) const;
bool get_a( const hashid_t& parmid, any& value ) const;

bool get_i(const string& parmname, int& value) const;
bool get_d(const string& parmname, double& value) const;
bool get_s(const string& parmname, string& value) const;
bool get_v(const string& parmname, numvec& value) const;
bool get_a(const string& parmname, any& value) const;

bool get_iptr(const hashid_t& parmid, int*& value);
bool get_dptr(const hashid_t& parmid, double*& value);
bool get_sptr(const hashid_t& parmid, string*& value);
bool get_vptr(const hashid_t& parmid, double*& value);
bool get_aptr(const hashid_t& parmid, any*& value);

bool get_iptr(const hashid_t& parmid, const int*& value ) const;
bool get_dptr(const hashid_t& parmid, const double*& value ) const;
bool get_sptr(const hashid_t& parmid, const string*& value ) const;
bool get_vptr(const hashid_t& parmid, const numvec*& value ) const;
bool get_aptr(const hashid_t& parmid, const any*& value ) const;

bool get_iptr(const string& parmname, int*& value );
bool get_dptr(const string& parmname, double*& value );
bool get_sptr(const string& parmname, string*& value );
bool get_vptr(const string& parmname, double*& value );
bool get_aptr(const string& parmname, any*& value );

bool get_iptr(const string& parmname, const int*& value ) const;
bool get_dptr(const string& parmname, const double*& value ) const;
bool get_sptr(const string& parmname, const string*& value ) const;
bool get_vptr(const string& parmname, const double*& value ) const;
bool get_aptr(const string& parmname, const any*& value ) const;

// type II getter function
int& get_i( const hashid_t& parmid );
double& get_d( const hashid_t& parmid );
string& get_s( const hashid_t& parmid );
any& get_a( const hashid_t& parmid );

const int& get_i( const hashid_t& parmid ) const;
const double& get_d( const hashid_t& parmid ) const;
const string& get_s( const hashid_t& parmid ) const;
const numvec& get_v( const hashid_t& parmid ) const;
const any& get_a( const hashid_t& parmid ) const;

int& get_i( const string& parmname );
double& get_d( const string& parmname );
string& get_s( const string& parmname );
any& get_a( const string& parmname );

const int& get_i( const string& parmname ) const;
const double& get_d( const string& parmname ) const;
const string& get_s( const string& parmname ) const;
const numvec& get_v( const string& parmname ) const;
const any& get_a( const string& parmname ) const;

bool has_i( const hashid_t& parmid ) const;
bool has_d( const hashid_t& parmid ) const;
bool has_s( const hashid_t& parmid ) const;
bool has_v( const hashid_t& parmid ) const;
bool has_a( const hashid_t& parmid ) const;

bool has_i( const string& parmname ) const;
bool has_d( const string& parmname ) const;
bool has_s( const string& parmname ) const;
bool has_v( const string& parmname ) const;
bool has_a( const string& parmname ) const;

// connect to other pointer
bool connect(const pointer_t& ptr);
bool is_connected_to( const pointer_t& ptr) const;
bool disconnect(const pointer_t& ptr);

iterator_t related_atom_begin() const;
iterator_t related_atom_end() const;
iterator_t related_bond_begin() const;
iterator_t related_bond_end() const;
iterator_t related_angl_begin() const;
iterator_t related_angl_end() const;
iterator_t related_tors_begin() const;
iterator_t related_tors_end() const;
iterator_t related_tor2_begin() const;
iterator_t related_tor2_end() const;
iterator_t related_oops_begin() const;
iterator_t related_oops_end() const;
iterator_t related_ptor_begin() const;
iterator_t related_ptor_end() const;
iterator_t related_resd_begin() const;
iterator_t related_resd_end() const;

range_t related_atoms() const;
range_t related_bonds() const;
range_t related_angls() const;
range_t related_torss() const;
range_t related_tor2s() const;
range_t related_ptors() const;
range_t related_resds() const;

int related_atom_number() const;
int related_bond_number() const;
int related_resd_number() const;
int related_angl_number() const;
int related_oops_number() const;
int related_tors_number() const;
int related_tor2_number() const;
int related_ptor_number() const;

bool has_related_atom(ptrpred_t pred) const;
bool has_related_bond(ptrpred_t pred) const;
bool has_related_resd(ptrpred_t pred) const;
bool has_related_angl(ptrpred_t pred) const;
bool has_related_tors(ptrpred_t pred) const;
bool has_related_tor2(ptrpred_t pred) const;
bool has_related_ptor(ptrpred_t pred) const;
bool has_related_oops(ptrpred_t pred) const;

bool get_related_atom(ptrpred_t pred, pointer_t& atom) const;
bool get_related_bond(ptrpred_t pred, pointer_t& bond) const;
bool get_related_resd(ptrpred_t pred, pointer_t& resd) const;
bool get_related_angl(ptrpred_t pred, pointer_t& angl) const;
bool get_related_tors(ptrpred_t pred, pointer_t& tors) const;
bool get_related_tor2(ptrpred_t pred, pointer_t& tor2) const;
bool get_related_ptor(ptrpred_t pred, pointer_t& ptor) const;
bool get_related_oops(ptrpred_t pred, pointer_t& oops) const;

pointer_t get_related_atom(ptrpred_t pred) const;
pointer_t get_related_bond(ptrpred_t pred) const;
pointer_t get_related_resd(ptrpred_t pred) const;
pointer_t get_related_angl(ptrpred_t pred) const;
pointer_t get_related_tors(ptrpred_t pred) const;
pointer_t get_related_tor2(ptrpred_t pred) const;
pointer_t get_related_ptor(ptrpred_t pred) const;
pointer_t get_related_oops(ptrpred_t pred) const;
```

4.6.2 Iterator

In practise, some times we need to iterate overs some contiguous objects. For example, we may want to iterate over all atoms of a molecule, or all related atoms of an atom. The first one can be archieved by using member function `absid_begin()` and `absid_end()` of class `component_t`. As the following code has shown:

```
    molecule_t mol;
    .....
    const component_t* atoms = mol.get_component(ATOM);
    vector<int>::iterator id_i = atoms->absid_begin();
    for( ; id_i != atoms->absid_end(); ++id_i )
    {
        pointer_t patom(mol, ATOM, *id_i);
        std::cout << "atom␣" << patom.absid() << "␣name␣" << patom.get_s(NAME) << std::endl;
    }
```

The second goal can be archived via similar code:

```
    molecule_t mol;
    .....

    const adjacency_t* a_a = mol.get_adjacency(ATOM, ATOM);
    vector<int>::iterator id_i = a_a->begin( 0 );
    for( ; id_i != a_a->end( 0 ); ++id_i )
    {
        pointer_t patom(mol, ATOM, *id_i);
        std::cout << "related␣atom␣" << patom.absid() << "␣name␣" << patom.get_s(NAME) << std::en
    }
```

The similarity of the two pieces of codes inspires us to develop class `iterator_t`, which can be considered as pointer of `pointer_t`. It also takes 3 arguments to declare, the molecule, the component ID and the iterator of type `vector<int>`. When dereference an `iterator_t`, it will return an instanse of `pointer_t`. The following is full list of member functions of class `iterator_t`.

```
  iterator_t( const molecule_t& mol, int cid, index_t index );
  iterator_t( const iterator_t& rhs );
  virtual ~iterator_t();
  iterator_t& operator=( const iterator_t& rhs );
  const index_t& getidx() const;
  pointer_t& operator*();
  pointer_t const& operator*() const;
  pointer_t* operator->();
  pointer_t const* operator->() const;
  iterator_t& operator++(void);
  iterator_t& operator--(void);
  iterator_t operator++(int);
  iterator_t operator--(int);
  iterator_t& operator+=( ptrdiff_t dif );
  iterator_t& operator-=( ptrdiff_t dif );
```

and as has shown before, class `molecule_t` and `pointer_t` both have some member functions returns an `iterator_t`. Thus, the above two piece of code can be rewritten as following using `iterator_t`:

```
    iterator_t ai = mol.atom_begin();
    for( ; ai != mol.atom_end(); ++i )
    {
        cout << "atom␣" << ai->absid() << "␣name␣" << ai->get_s(NAME) << std::endl;
    }
```

and:

```
    pointer_t center( mol, ATOM, 0 );
    iterator_t ai = center.related_atom_begin();
    for( ; ai != center.related_atom_end(); ++i )
    {
        cout << "atom␣" << ai->absid() << "␣name␣" << ai->get_s(NAME) << std::endl;
    }
```

4.6.3 Range

Class `range_t` is provides for those who would rather use an array to access an object than use an iterator. It takes two iterator to declare a `range_t`, and it works just list an object array. The following is the member functions of `range_t`,

```
  range_t( const iterator_t& begin, const iterator_t& end )
  iterator_t begin() const;
  iterator_t end()   const;
  int size() const;
  pointer_t at( int id ) const;
  pointer_t operator[]( int id ) const;
```

and the example code can be rewriten as the following code using `range_t`:

```

    range_t atoms = mol.atoms();
    for( int i=0; i < atoms.size(); ++i )
    {
        std::cout << "atom␣" << atoms[i].absid() << "␣name␣" << atoms[i].get_s(NAME) << std::endl;
    }
```

and:

```

    pointer_t center(mol, ATOM, 0);
    range_t atoms = center.related_atoms();
    for( int i=0; i < atoms.size(); ++i )
    {
        std::cout << "atom␣" << atoms[i].absid() << "␣name␣" << atoms[i].get_s(NAME) << std::endl;
    }
```

Chapter 5

Introduction to gleap

In this chapter, we will introduce gleap about its structure, and give two examples to illustrate how to add new commands to gleap.

5.1 Code orgnization and structure

The code of gleap located in three directories. In directory *gleap/mortsrc/guilib* there are classes shared by *gleap* and *sleap*; in directory *gleap/leapsrc* there are classes and entry points of *gleap* and *sleap*; in directory *gleap/plugins* there are classes which implements all the commands in leap.

As for the internal structure, gleap uses a MVC (Model-View-Control) pattern which is used almost by all UI programs. It has the big advantage of dividing the whose system into three less-coupled components. The model part hold all the data (sometime called content or document), the view part is in charge of system output and the control part takes user input and give instructions to model and view. In fact, a computer can be thought as a MVC system. The model is memory, view is monitor, and control is CPU.

In gleap, the three components has different names. They are now content (model), drawing (view) and control.

First of all, content as the model part of gleap, is not a class but a function. it is declared in file *gleap/mortsrc/guilib/mainwin.hpp*:

```
    database_t& content();
```

and simply returns a reference of type mort::database_t. It is because database_t already proivded all the interfaces we need for a model class, so we just declare a static database_t inside function content(), and return its reference.

The class drawing_t as the view part of gleap's MVC pattern, is defined in file *gleap/mortsrc/guilib/drawing.hpp*, and has the following member functions:

```
void add( const shared_ptr< graphic_i >& graphic );

bool has( const string& gname ) const;

void remove( const string& graphic );

void init();

void repaint();

void resize( int width, int height );

virtual void glbegin() = 0;
virtual void glend() = 0;

virtual void expose() = 0;
```

most of these functions is manipulating shared pointer of class *graphic_i*, which is the base class for all the graphic objects. It is defined with the following virtual functions:

```
  virtual string name() const = 0;

  virtual void paint() = 0;
```

The first one returns the name of a graphic, the second one using OpenGL routines to paint the object. Currently we have the two classese inherited from graphic_i and could then be put into drawing. Class molecule_graphic shows a molecule and class ribbon_graphic shows the secondary structure of a protein.

Class drawing_t works as a container of graphic_i. It simply calls the function *paint()* of each graphic it contains when display is request. Thus, to update display windw, users need to add, remove or modify graphics. For instance, in order to display a molecule on screen, you will need construct a molecule_graphic first, then add it to drawing.

The other three member functions of drawing is *glbegin()*, *glend()*, *expose()*. they are defined as virtual functions, since different windwo systems has their own methods to initialize an OpenGL environment. Currently there are two classes inherited from *drawing_t*, and are used by different executables. Class n̂ull_drawing is used by text based program *sleap*, so its *glbegin()*, *glend()* and *expose()* do nothing. Class *gtkarea_t* is used by GTK based program *gleap*, and it uses functions from library *gtkglext* for OpenGL environment initialzation. Users will need to implement their own drawing classes if they want to extent gleap to other system such as Qt and Motif.

Class control_t in charge of the whole system is the most important class in gleap. It is defined in file *gleap/-mortsrc/guilib/control.hpp* with the following member functions:

```
static void insert( const string& name, command_i* ptr );

static bool run( const string& command );

static std::map< string, command_i* >::iterator begin();

static std::map< string, command_i* >::iterator end();
```

and control has a data member *g_commands*, declared as the following:

```
    static map< string, command_i* > g_commands;
```

it works as a dictionary which map command's name to its pointer. *command_i* is another important class which will be discussed later.

Member function *insert()* is used for command registration. It add or update an entry in control's *g_commands*.

Member function *run()* is the interface for running a command. For a given command line, *run()* parse it into command name and arguments, then find the command in *g_commands*, and then execute it with the parsed arguments.

class *command_i* defined in file *gleap/mortsrc/guilib/command.hpp* is the base type that all command class should inherit from. It has the following member function:

```
  command_i();

  command_i( const string& name );

  virtual ˜command_i();
```

```
virtual bool exec() = 0;

virtual void undo() = 0;

virtual shared_ptr<command_i> clone(const vector<string>& args)const=0;
```

Three of these member functions are purely virtual, means *ommand_i* haven't implemented these function, and they need to be implemented in the derived classes. Among them, *exec()* is for command execution, while *undo()* is about to undo the effect made by the execution. The member function *clone()* worths a little more discussion, every time *control_t* execute a command, it create a new command object by calling *clone()* with the input arguments. then call *exec()* with the newly created command object. All the information about this execution will be stored inside the command object which is put into a command array named *history*. The *history* is used to support the undo-redo mechanism of gleap. At the time when we write this manual, the undo-redo mechanism have not been implemented, but it should be available very soon. Another member function need to noted is constructor *command_i(const string& name)*, which calls member function *insert()* of *control_t* to register the command in command dictionary of control.

There are some other classes which is important but is not mentioned.

class *console_t*, which is used for processing keyboard events. It has the following memeber functions:

```
/// handler for key up pressed
void on_up();

/// handler for key tab pressed
void on_tab();

/// handler for key down pressed
void on_down();

/// handler for key enter pressed
bool on_enter();
```

to handle different keyboard event, among them *on_enter* is the most important one since it get a command line and call *run* of *control_t* to execute the command.

class leaplog_t is used for log. It has the following member functions:

```
  void leaplog_t::putline( const string& line )

  bool leaplog_t::getline( string& line )
```

usually command should put their messages to leaplog via putline, these messages will be retrived and put to screen by control_t after command execution.

5.2 Adding new commands

In this section, we will introduce about how to add new command to gleap. We will illustrate hwo to do that by showing some examples. Before that, it is important that users re-run the configure of gleap with option "with-loadlib", which will cause a command "loadlib" to be built in gleap. This command allow users to load new command into leap from a shared library, to avoid modifying leap source code directly.

5.2.1 Hello

As our first example, we illustrate a simple command hello, which takes one argument and print out a grating message on the screen.

The full source code of hello is listed here, you can also found it under directory *gleap/example/leap/hello/*:

```
#include <object.hpp>
#include <guilib.hpp>

using namespace mort;

class hello_command : public command_i
{
public:

  hello_command( );

  hello_command( const string& target );

  virtual ~hello_command( );

  virtual bool exec( );

  virtual void undo( );

  virtual shared_ptr< command_i > clone( const vector< string >& args ) const;

private:

  string m_target;

};


hello_command::hello_command( )
  : command_i( ``hello'' )
{
}

hello_command::hello_command( const string& target )
  : m_target( target )
{
}

hello_command::~hello_command( )
{
}

bool hello_command::exec( )
{
  leaplog_t::putline( ``Hello, '' + m_target + ``!'' );
  leaplog_t::putline( ``How are you?'' );
}

void hello_command::undo( )
{
}

shared_ptr<command_i> hello_command::clone(const vector<string>& args) const
{
  if( args.size() != 2 )
  {
    throw logic_error( ``Error: wrong number of argument'' );
  }

  return shared_ptr< command_i >( new hello_command( args[1] ) );
}

hello_command g_hello_command;
```

the code itself is self-explainary. Noted here several points:

1. the statement "*hello_command g_hello_command;*" is important since it causes the command itself be registered in control_t's command dictionary with the name "hello".

2. in function *clone()*, there is a test if args's size if 2, the first element of args are always the command name. Acturally, one command class can be register more one time in command dictionary with different, they can have differetn behavior by args[0].

The Makefile used for hello is also listed here:

```
libhello.la : hello.o
        g++ -o libhello.la    \
                    hello.lo -rpath /usr/local/lib   \
                    -version-info 0:0:0 -release 1.0 \
                    -lmort -L../../../mortsrc

hello.o : hello.cpp
        g++ -o hello.lo -c hello.cpp -I../../../mortsrc \
                    -I../../../freelib
```

which will generate a libtool library libhello.la generated, the tag *-rpath /usr/local/lib -version-info 0:0:0 -release 1.0* in the linking command is important, without which only static library will be created, and static library is not allow to be load into memory at run-time.

if everything fine you should a libtool library libhello.la under your current directory, you just need to load it in the following steps:

1. start sleap from your current directory, (gleap works too but since we are not going to use any graphic feature here, we would rather use sleap).

2. try to run command *help* which will give you a list of commands, make sure *loadlib* is in the list.

3. try typein *loadlib hello*, if everything goes smoothly, you should get the following information *library commands have been loaded from libhello.la*.

4. try *help* again, you should have *hello* in your command list

5. input command *hello world*, if you get the the following greeting message:

```
   Hello, world!
   How are you?
```

then congratulations, you have finished you very first gleap command.

5.2.2 makeles

In last subsection, we introduced the development of command hello, which is acturally useless except demonstrating some basic concept. In this subsection we will develop a more useful command: makeles, which will make multiple copies on selected region of a molecule, as we introduced before.

Before we start the development of makeles, let's explain two basic operations of gleap programming, getting molecule from cotent and putting graphics to drawing.

As we have mentioned, function *content()* refer to a database which has all informations stored in, including the molecules. In principal, we can get a molecule using *database_t* member function *get()*. The problem is *get()* return a shared pointer of type *entity_t*, you still need to convert is to *molecule_t*, using *dynamic_pointer_cast*. There is a function *get_unit()* defined to save you from all this problems, the declaration can be found in file *gleap/mortsrc/guilib/contain.hpp*, as the following:

```
    molecule_t& get_unit( const string& name );
```

you can even get a residue by calling *get_resd()* with a string with format "xxx.nnn", here xxx is molecular name, and nnn is residue number. Similarly, function *get_atom()* allow you to get an atom by giving a string "xxx.nnn.yyy", here yyy is atom's name, again we are assuming there are not two atoms in a residue with same names.

Another important operation we need is displaying a molecule on screen, this is done by construct a moelcular graphic then put it into *drawing()*. Class *molecule_graphic* has the following constructor defined:

```
  molecule_graphic( const molecule_t& mol, int style )
```

Here style could either be LINE or BALLSTICK, and the graphic can be add to *drawing()* using its member function *add()*.

Here we list the source code of class *makeles_command*, you can also find it under directory *gleap/example/-makeles/*. It calls function *make_copy*, whose code has been listed in chapter 2, so we are not going to repeat it here.

```
class makeles_command : public command_i
{
public:

  makeles_command();

  makeles_command(const string& dst,const string& src,
                  const string& mask,int ncopy );

  virtual ˜makeles_command( );

  virtual bool exec( );

  virtual void undo( );

  virtual shared_ptr<command_i> clone(const vector<string>& args)const;

private:

  string m_dst;

  string m_src;

  string m_mask;

  int m_ncopy;

};

makeles_command::makeles_command( )
  : command_i( "makeles" )
{
}

makeles_command::makeles_command( const string& dst, const string& src,
                                  const string& mask, int ncopy  )
  : m_dst( dst ), m_src( src ), m_mask( mask ), m_ncopy( ncopy )
{
}

makeles_command::~makeles_command( )
{
}

bool makeles_command::exec( )
{
  shared_ptr< molecule_t > pdst;
  pdst = make_copy( get_unit( m_src ), m_mask, m_ncopy );
  content().set( m_dst, pdst );

  shared_ptr< graphic_i > mgraph(new molecule_graphic(*pdst,LINE));
  drawing()->add( mgraph );
}

void makeles_command::undo( )
{
}

shared_ptr<command_i> makeles_command::clone(const vector< string >& args) const
{
  if( args.size() != 5 )
  {
    throw logic_error( ``Error: wrong number of argument'' );
  }

  int ncopy = atoi( args[4].c_str() );

  if( ncopy <= 0 )
  {
    throw logic_error( ``Error: wrong copy number '' + args[4] );
  }

  return shared_ptr<command_i>(new makeles_command(args[1],args[2],
                                                   args[3], ncopy) );
}

makeles_command g_makeles_command;
```

the command takes 4 arguments:

dst	the name of the new molecule
src	the name of the input molecule
mask	select the region to be copied
ncopy	number of copies to be made

Note gleap support two grammers. For standard space seperated command line, the first word will be considered as command name, the rest words are arguments; If a command line containing a "=" sign is encoutered, the word right after the "=" is taken as the command name, while the word before it is the first argument. Thus the next two command is equivalent:

```
    makeles dst src mask ncopy
    dst = makeles src mask ncopy
```

We prefer the second one, which is more meaningful to us.

Hence we list a short script for leap, which load the library we just built, read in a molecule make 4 copies of the first residue, then save it to amber prmtop file:

```
  #!/bin/csh
  cat > leap.in << EOF
```

```
loadlib makeles
trip = loadmol2 trip.mol2
les  = makeles trip :1 4
savemol2 les les.mol2
loadamberparams parm99.dat
saveamberparm les les.top les.xyz
quit

EOF

\$AMBERHOME/src/gleap/leapsrc/sleap < leap.in
```

The file *trip.mol2* mentioned in the script can be found under directory *gleap/example/leap/makeles/*, along with the script itself named *Run.makeles*