```
#include "small_matrix.hpp"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <valarray>

typedef Math::tMatrix<3> Matrix;
typedef Math::tVector<3> Vector;

class Reflection {
  public:
    Vector hkl;
    float intensity;
    float sigma;
    float inverse_sqr_sigma;    // accumulates 1/sqare(sigma)
    float size;                 // and n, for equivalent reflections
    bool centric;
    bool absent;
    float phase_shift;

    Reflection (){
        size=1;                 // n=1 for each unmerged reflection
        centric = false;
```

```
        absent  = false;
    }

    // create a reflection from hkl, I and sigma etc
    Reflection (Vector v, float i, float s, bool ab=false,
            bool cen=false, float shft = 0.0) {
        hkl       = v;
        intensity = i;
        sigma     = s;
        size      = 1.0;
        inverse_sqr_sigma = 1.0 / (s * s);
        absent    = ab;
        centric   = cen;
        phase_shift = shft;
    }

    // adds another reflection to this one if equivalent and return true
    // if not equivalent, just return false
    bool add(const Reflection & another_refl) {
        if ( this->hkl == another_refl.hkl ) {
            this->intensity         += another_refl.intensity;
            this->inverse_sqr_sigma += another_refl.inverse_sqr_sigma;
            this->size += 1.0;
            this->centric = (this->centric || another_refl.centric);
            this->absent  = (this->absent || another_refl.absent);
            return true;
        }
```

```
        return false;
    }

    float merged_sigma() {
        return   1.0 / ( sqrt (inverse_sqr_sigma));
    }

    float merged_intensity() {
        return   intensity / size;
    }

    float intensity_over_sigma() {
        return   intensity / (size * sqrt (inverse_sqr_sigma));
    }

}; // Reflection


// comparison of reflections
bool operator < (const Reflection& lhs, const Reflection& rhs) {
    return (lhs.hkl < rhs.hkl);
}

// equality of reflections
bool operator == (const Reflection& lhs, const Reflection& rhs) {
    return (lhs.hkl == rhs.hkl);
}
```

```
// pretty printing of reflections
std::ostream& operator<<(std::ostream& os, Reflection& ref) {
   using namespace std;
   os<<setiosflags(ios::fixed);
   os<<setprecision(3);
   os<<"Reflection "<<ref.hkl<<"  I/sigma = "
        <<setw(8)<<ref.intensity_over_sigma();
   return os;
}

// A function object initialized with a lists of symops
// (rotational, translational) Applied to each reflection,
//     it translates each input hkl to the reduced hkl and returns it.
struct HKLReducer {
    std::vector< Matrix > Rt;                       // transpose of rotational
    std::vector< std::valarray<float> > Tr;     // translational
    Vector hkl_m;
    int  max;

    HKLReducer( const std::vector< Matrix > &rot ,
                const std::vector< std::valarray<float> > &trans) {
        Rt = rot;
        Tr = trans;
        max = (rot.size() * 2)-1;
    }
```

```
    // this is where the magic happens
    Reflection operator() (const Reflection& ref) {
        const Vector& prime_hkl = ref.hkl;
        bool centric=false;
        bool absent = false;
        float phase_shift;

        std::vector< Vector > list_of_equivalents;
        list_of_equivalents.push_back(prime_hkl);     // store prime HKL and
        list_of_equivalents.push_back(-prime_hkl);    // its friedel mate

        // loop over rotational symops Rt_m and generate equivalent hkl_m
        for (int m=1; m< Rt.size(); ++m) {
            hkl_m = Rt[m] * prime_hkl;
            list_of_equivalents.push_back(hkl_m);    // add hkl_m and
            list_of_equivalents.push_back(-hkl_m);  // its friedel mate to list

            // identify systematic absences
            if (hkl_m == prime_hkl) {
                phase_shift = (
                      (Tr[m])[0] * prime_hkl(0)
                    + (Tr[m])[1] * prime_hkl(1)
                    + (Tr[m])[2] * prime_hkl(2)
                );
                if (std::abs(phase_shift - round(phase_shift)) > 0.001)
                   absent = true;
            } else if (hkl_m == -prime_hkl) {
```

```
                centric = true;
            }
        }

        // The maximum hkl in the list represents the bunch
        sort(list_of_equivalents.begin(),list_of_equivalents.end());
        return Reflection(list_of_equivalents[max],
                ref.intensity, ref.sigma, absent, centric, phase_shift);
    }
};

// processor for counting and printing reflections based on
// absent/centric etc.
class printHKL{
  public:
    int absent;
    int centric;
    int rest;

    printHKL() {
        rest = centric = absent = 0;
    }

    void operator() (Reflection &r){
        if (r.absent) {
            ++absent;
            std::cout<<r<<std::endl; // print only syst. absent
```

```
        } else if (r.centric) {
            ++centric;
            ++rest;
        } else ++rest;
    }

    void complete() {
        std::cout<<std::endl<<rest<<" unique reflections of which "
            <<centric<<" are centric"<<std::endl;
        std::cout<<absent<<" systematic absences"<<std::endl;
    }

    ~printHKL() {
       complete();
    }

} my_printer;
```

```
// START
int main () {
    std::ifstream data_file ("in");
    std::string junk_string;
    int number_of_symops=6;   // symops to read from file

    std::vector< Matrix >           list_of_Rt;
    std::vector< std::valarray<float> >  list_of_Tr;
    Matrix                Rt(false);
    std::valarray<float> Tr(3);

    // Skip header
    for (int i=0 ; i < 3 ; ++i)
        getline (data_file,junk_string);

    // Read symops and store the transposes;
    for (int i=0 ; i < number_of_symops ; ++i) {
        for (int k=0; k < 9; ++k) {
            data_file >> Rt(k % 3, k / 3); // swap i and j for transpose
        }
        data_file >>Tr[0]>>Tr[1]>>Tr[2];
        list_of_Rt.push_back( Rt );
        list_of_Tr.push_back( Tr );
    }

    // let the reducer know which symops to use.
    HKLReducer my_hkl_reducer(list_of_Rt, list_of_Tr);
```

```
    std::vector<Reflection> data_list;
    Vector tmp_hkl;
    float tmp_i, tmp_s;

    // read the rest of the input file and process one reflection at a time
    while (!data_file.eof()) {
        // read in line of data: h, k, l, int, sigma
        data_file>>tmp_hkl(0)>>tmp_hkl(1)>>tmp_hkl(2)>>tmp_i>>tmp_s;

        // transform Reflection to asym unit and store in data_list
        data_list.push_back(
            my_hkl_reducer(
                Reflection(tmp_hkl, tmp_i, tmp_s)
            )
        );
    }
    sort(data_list.begin(), data_list.end());

    // loop through data_list, add up multiple entries and
    // transfer into new vector

    std::vector<Reflection> merged_data;
    merged_data.push_back( data_list[0] );  // store first element before loop
    int my_pos = 0;
    for( int i=1; i < data_list.size(); ++i) {
        if ( ! merged_data[my_pos].add( data_list[i] ) ) {
```

```
            ++my_pos;
            merged_data.push_back( data_list[i]);
        }
    }

    for_each(merged_data.begin(), merged_data.end(), my_printer);
    my_printer.complete();
}

//END
```

```
/* ------------ important snip from small_matrix.hpp----------

// overload mult operator for multiplying matrix and vector
const Vector operator * (const Matrix& lhs,const Vector& rhs) {
    Vector tmp;
    for ( int i = 0; i < 3; ++i) {
```

```
        tmp(i) = 0;
        for ( int j = 0; j < 3; ++j)
            tmp(i) += static_cast<int>( lhs(i,j) * rhs(j) );
    }
    return tmp;
}

// a comparison operator is needed to sort a container of Vectors
bool operator < (const Vector& lhs, const Vector& rhs) {
    if (lhs(0) < rhs(0)) {
        return true;
    } else if (lhs(0) == rhs(0)) {
        if (lhs(1) < rhs(1)) {
            return true;
        } else if (lhs(1) == rhs(1)) {
            if (lhs(2) < rhs(2)) {
                return true;
            } else return false;
        } else return false;
    } else return false;
}

// test element-wise equality of vectors
bool operator == (const Vector& lhs, const Vector& rhs) {
    if (lhs(0) == rhs(0))
        if (lhs(1) == rhs(1))
            if (lhs(2) == rhs(2))
                return true;
```

```
    return false;
}
*/
```