```cpp
/**
 * \file    sort_reflections.cpp
 * \date    16/08/2005
 * \author  Tim Gruene
 * read in a file with header (3 + n(symops) lines) and sort reflections,
 * first by h, then by k, then by l
 */

#include <algorithm>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>
#include <sstream>
#include <string>

// definition of a useful struct
struct Symop
{
        int R[3][3];
        int T[3];
};

class Reflection
{
    private:
        int h_, k_, l_;
        float I_, sigI_;
        bool absent;
    public:
        Reflection(int h, int k, int l, float I, float sigI):
            h_(h), k_(k), l_(l), I_(I), sigI_(sigI), absent(false){}
        ~Reflection(){}

        // retrieve data members
        int   h() const { return h_;}
        int   k() const { return k_;}
        int   l() const { return l_;}
        float I() const { return I_;}
        float sigI() const { return sigI_;}

        // equality operator -- based on indices only
        bool operator==(const Reflection& r) const
        { return ( (h_ == r.h_) && (k_ == r.k_) && (l_ == r.l_));}

        // comparison operator
        inline bool operator<(const Reflection&) const;

        // multiplication with Symop
        inline Reflection operator*(const Symop&) const;

        //inline Reflection operator*(int) const;
        //! negation operator
        inline Reflection operator-() const;

        // set absence flag
        void set_absence() { absent=true;}

        // returns sym.-equivalent with maximal index
        Reflection max_equivalent(std::vector<Symop>&) const;

        // check absence
        friend bool is_absent(const Reflection& r) { return r.absent;}

        // for printing
        friend std::ostream& operator<<(std::ostream& os, const Reflection&);
```

```cpp
};

typedef std::vector<Reflection> Reflexes;

// forward declarations of functions called from main
void  usage();
int   read_reflections(char *, std::vector<Symop>&, std::vector<Reflection>&);
int   merge_reflections(const Reflexes&, Reflexes&);
int   remove_sys_abs(const std::vector<Symop>&,Reflexes&);
int   sys_absences(const std::vector<Symop>&,Reflexes&);
float Rint(const std::vector<Symop>&, const Reflexes&, const Reflexes&);


int main(int argc, char* argv[])
{
    std::vector<Reflection> reflexes, refl_merged;
    std::vector<Symop>      symops;
    std::vector<Reflection>::iterator it;

    if ( argc < 2 )
    {
        usage();
        return -1;
    }

    if (read_reflections(argv[1], symops, reflexes))
    {
        std::cerr << "An error occurred while reading " << argv[1] << '\n';
        return EXIT_FAILURE;
    }

    std::cout << "Standardising indices.\n";
    for ( it = reflexes.begin(); it != reflexes.end(); it++)
    {
        *it = it->max_equivalent(symops);
    }

    std::cout << "Sorting reflections.\n";
    std::sort(reflexes.begin(), reflexes.end());

    std::cout << "Merging symmetry equivalents.\n";
    merge_reflections(reflexes, refl_merged);

    std::cout << "Analysing systematic absences.\n";
    sys_absences(symops, refl_merged);

    std::cout << "Number of reflections: " << reflexes.size() << '\n';

    std::cout << "After merging, there are " << refl_merged.size()
        << " unique reflections.\n";

    std::cout << "R(int)  = "
        << std::fixed << std::setprecision(4)
        << Rint(symops, reflexes, refl_merged) << '\n';

    return 0;
}

// compare hkl indices of two reflections
bool Reflection::operator<(const Reflection& other)const
{
    if ( h_ < other.h_ ) return true;
    else if ( h_ > other.h_) return false;
    else        // h1 == h2
    {
        if ( k_ < other.k_) return true;
        else if ( k_ > other.k_) return false;
        else    // k1 == k2
        {
            if ( l_ < other.l_ ) return true;
            else if ( l_ > other.l_) return false;
        }
    }

    // h1 == h2 && k1 == k2 && l1 == l2
    return  false;
}

// finds the equivalent with maximal indices
Reflection Reflection::max_equivalent(std::vector<Symop>& symops) const
{
    Reflection max_refl = *this;
    std::vector<Symop>::const_iterator it;

        for ( it = symops.begin(); it != symops.end(); it++)
        {
            Reflection sym_refl = ((*this)* (*it));

            if ( max_refl < sym_refl )
            {
                max_refl = sym_refl;
            }
            else if ( max_refl < -sym_refl)
            {
                max_refl = -sym_refl;
            }
        }
    return max_refl;
}

Reflection Reflection::operator*(const Symop& symop) const
{
    Reflection produkt(*this);

    produkt.h_ = h_*symop.R[0][0] + k_*symop.R[1][0] + l_*symop.R[2][0];
    produkt.k_ = h_*symop.R[0][1] + k_*symop.R[1][1] + l_*symop.R[2][1];
    produkt.l_ = h_*symop.R[0][2] + k_*symop.R[1][2] + l_*symop.R[2][2];

    return produkt;
}

/*
Reflection Reflection::operator*(int m) const
{
    Reflection r(*this);
    r.h_ = m*this->h_;
    r.k_ = m*this->k_;
    r.l_ = m*this->l_;

    return r;
}
*/

/**
 * returns a reflex with negated indices
 */
Reflection Reflection::operator-() const
{
    Reflection neg(*this);
    neg.h_ = -neg.h_;
    neg.k_ = -neg.k_;
    neg.l_ = -neg.l_;

    return neg;
```

```
}

std::ostream& operator<<(std::ostream& os, const Reflection& r)
{
    os << std::fixed
        << std::setw(6) << r.h_
        << std::setw(6) << r.k_
        << std::setw(6) << r.l_
        << std::setw(10) << std::setprecision(3) << r.I_
        << std::setw(10) << std::setprecision(3) << r.sigI_;
    return os;
}



void usage()
{
    std::cout << "Usage: siena <filename>.\n";
}

/**
 * read a list of reflections from filename. Format must be:
 * line 1:  title/comment
 * line 2:  cell ( a b c alpha beta gamma)
 * line 3:  n symops ( n = number of symops)
 * line 4 - 4+n: symops
 */
int  read_reflections(char * filename,
        std::vector<Symop>& symops, std::vector<Reflection>& refls)
{
    std::ifstream input(filename);
    std::string   line;                 // for getline
    int   num_symops;

    if (! input.is_open())
    {
        std::cout << "Could not open file " << filename << '\n';
        return -1;
    }

    // skip first two lines
    std::getline(input, line);
    std::getline(input, line);

    // get number of symops
    std::getline(input, line);
    std::istringstream num_symops_stream(line);

    num_symops_stream >> num_symops;

    // read in symops
    for ( int i = 0; i < num_symops; i++)
    {
        std::getline(input, line);
        std::istringstream symop_stream(line);

        Symop symop;
        float tx, ty, tz;

        for ( int j = 0; j < 3; j++)
            for ( int k = 0; k < 3; k++)
                symop_stream >> symop.R[j][k];
        symop_stream >> tx >> ty >> tz;

        symop.T[0] = int ( 0.5+12*tx);
        symop.T[1] = int ( 0.5+12*ty);
        symop.T[2] = int ( 0.5+12*tz);
        symops.push_back(symop);
    }

    // now read in reflections
    refls.clear();

    while ( true )
    {
        int h, k, l;
        float I, sigI;
        input >> h >> k >> l >> I >> sigI;
        if(input.eof()) break;
        refls.push_back(Reflection(h,k,l,I,sigI));

        /*
        std::istringstream refl_stream(line);

        refl_stream >> h >> k >> l >> I >> sigI;
        refls.push_back(Reflection(h,k,l,I,sigI));
        // get next line
        std::getline(input, line);
        */
    }

    return 0;
}

/**
 * Merges list of Reflections unmerged based on the list of symmetry operators
 * and puts the merged list into merged
 */
int  merge_reflections(const Reflexes& unmerged, Reflexes& merged)
{
    Reflexes::const_iterator it;

    for ( it = unmerged.begin(); it != unmerged.end(); it++)
    {
        double sumI      = 0.0;
        double sum_sigma = 0.0;
        int    num_sym_equiv = 0;

        Reflection reference(*it);

        while ( reference == *it )
        {
            sumI         += it->I();
            sum_sigma    += 1.0/(it->sigI() * it->sigI());
            ++num_sym_equiv;
            ++it;
        }
        merged.push_back(Reflection(reference.h(), reference.k(), reference.l(),
                                sumI/num_sym_equiv, 1.0/std::sqrt(sum_sigma)));
        // rewind by one
        --it;
    }

    return 0;
}

/**
 * checks merged for systematically absent reflections and prints I/sigI for
 * these reflections. They are removed from the list. Also counts number of
 * centric reflections
 * \param   sympops list of symmetry operators
 * \param   merged  list of merged reflexes
 */
```

```cpp
int sys_absences(const std::vector<Symop>& symops,Reflexes& merged)
{
    Reflexes::iterator it_r;
    std::vector<Symop>::const_iterator it_sym;
    int num_sys_abs(0);
    int centric_reflections(0);
    bool is_centric;

    for ( it_r = merged.begin(); it_r != merged.end(); it_r++)
    {
        is_centric = false;
        for ( it_sym = symops.begin()+1; it_sym != symops.end(); it_sym++)
        {
            Reflection equiv = (*it_r)*(*it_sym);
            if ( equiv == (*it_r) )
            {
                if ( (!it_sym->T[0]) && (!it_sym->T[1]) && (!it_sym->T[2]) )
                {
                        continue;
                }
                else
                {
                    int shift = (equiv.h() * it_sym->T[0] +
                                 equiv.k() * it_sym->T[1] +
                                 equiv.l() * it_sym->T[2]);
                    if ( shift%12 )  // the remainder of division
                    {
                        it_r->set_absence();
                        std::cout << "Reflection " << *it_r
                            << " syst. abs., I/sigI: "
                            << std::fixed
                            << std::setw(7)
                            << std::setprecision(2)
                            << it_r->I()/it_r->sigI()
                            << '\n';
                    break;
                    }
                }
            }
            else                         // indices are not identical
            {
                if ( equiv == -(*it_r) ) // check for centric reflex
                {
                    is_centric = true;
                }
                continue;
            }
        }
        if ( is_centric ) ++centric_reflections;
    }

    // Now let's remove absent reflections from the list
    Reflexes::iterator it_remove;
    it_remove = std::remove_if(merged.begin(), merged.end(), is_absent);
    merged.erase(it_remove, merged.end());

    std::cout << "Number of centric reflections: " << centric_reflections
        << std::endl;
    return (num_sys_abs);
}

/**
 * Calculate R_int for list unmerged, <I> is taken from merged
 * \param   symops  list of symmetry operators
 * \param   unmerged list of sorted but unmerged reflections
 * \param   merged   list of merged reflections ( for <I>
 */
```

```cpp
float Rint(const std::vector<Symop>& symops, const Reflexes& unmerged,
           const Reflexes& merged)
{
    Reflexes::const_iterator it;
    Reflexes::const_iterator it_Imean;
    double R_int (0.0);
    double I_total(0.0);

    it_Imean = merged.begin();

    for ( it = unmerged.begin(); it != unmerged.end(); it++)
    {
        int    num_sym_equiv(0);
        float  Imean;
        double r_int (0.0);
        double i_total(0.0);

        Reflection test(*it);

        // we could also simply increase it_Imean after each while-loop
        //it_Imean = std::find(it_Imean, merged.end(), test);
        Imean    = it_Imean->I();
        if ( ! (*it_Imean == *it)) continue;
        ++it_Imean;

        while (test == *it && it != unmerged.end() )
        {
            r_int   += std::abs(it->I() - Imean);
            i_total += it->I();
            ++num_sym_equiv;
            ++it;
        }
        // only consider reflections with more than one symmetry mate
        if (num_sym_equiv > 1)
        {
            R_int   += r_int;
            I_total += i_total;
        }
        // while loop went one too far
        --it;
    }

    return  (R_int / I_total);
}
```