```
//
//  range.h
//  SimpleCacheSim
//
//  Created by Wade Tregaskis on 7/9/2005.
//
//  Copyright (c) 2005, Wade Tregaskis.  All rights reserved.
//  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
//    * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
//    * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other
materials provided with the distribution.
//    * Neither the name of Wade Tregaskis nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior
written permission.
//  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL
, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF TH
E USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


#include <list>
#include <iostream>

#include <errno.h>


template<typename _Tp> class range;

template<typename _Tp> class rangeIterator {
private:
    typename std::list< std::pair<_Tp, _Tp> >::const_iterator iter, stopper;
    _Tp positionWithinIter;

    friend class range<_Tp>;

public:
    typedef size_t size_type;

    explicit rangeIterator() : positionWithinIter(0) {}
    rangeIterator(typename std::list< std::pair<_Tp, _Tp> >::const_iterator theIter, typename std::list< std::pair<_Tp, _Tp> >::const_iterator theStopper, size_type
position) : iter(theIter), stopper(theStopper), positionWithinIter(position) {}
    rangeIterator(const rangeIterator<_Tp> &other) : iter(other.iter), stopper(other.stopper), positionWithinIter(other.positionWithinIter) {}

    const _Tp operator*() const {
        if (iter != stopper) {
            return iter->first + positionWithinIter;
        } else {
            static const _Tp zoiks = 0;
            return zoiks;
        }
    }

    const _Tp* operator->() const {
        return &(operator*());
    }

    rangeIterator<_Tp>& operator++() {
        if (iter != stopper) {
            if (positionWithinIter >= (iter->second - iter->first)) {
                ++iter;
                positionWithinIter = 0;
            } else {
                ++positionWithinIter;
            }
        }

        return *this;
    }

    rangeIterator<_Tp> operator++(int) {
        rangeIterator<_Tp> result = *this;

        operator++();

        return result;
    }

    bool operator==(const rangeIterator<_Tp> &other) const {
        return ((iter == other.iter) && (stopper == other.stopper) && ((iter == stopper) || (positionWithinIter == other.positionWithinIter)));
    }

    bool operator!=(const rangeIterator<_Tp> &other) const {
        return ((iter != other.iter) || (stopper != other.stopper) || ((iter == stopper) && (positionWithinIter != other.positionWithinIter)));
    }

    void operator=(const rangeIterator<_Tp> &other) {
        iter = other.iter;
        stopper = other.stopper;

        positionWithinIter = other.positionWithinIter;
    }
};


template<typename _Tp> std::ostream& operator<<(std::ostream& stream, const range<_Tp> &r);


template<typename _Tp> class range {
private:
    typename std::list< std::pair<_Tp, _Tp> > _range;

    friend std::ostream& operator<< <_Tp>(std::ostream& stream, const range<_Tp> &r);

public:
    typedef rangeIterator<_Tp> iterator;
    typedef size_t size_type;
```

```
explicit range() {}
range(const _Tp &initialValue) { _range.insert(_range.begin(), initialValue); }
range(const iterator &other) : _range(other._range) {}
~range() {}

iterator& operator=(const iterator &other) {
    _range = other._range;
}

iterator& operator=(const _Tp &value) {
    _range.clear();
    _range.insert(_range.begin(), value);
}

iterator begin(void) const {
    return iterator(_range.begin(), _range.end(), 0);
}

iterator end(void) const {
    return iterator(_range.end(), _range.end(), 0);
}

bool empty() const { return _range.empty(); }

size_type size() const {
    typename std::list< std::pair<_Tp, _Tp> >::const_iterator iter, stopper;
    size_type total = 0;

    for (iter = _range.begin(), stopper = _range.end(); iter != stopper; ++iter) {
        total += iter->first - iter->second + 1;
    }

    return total;
}

size_type max_size() const { return 1 << (sizeof(_Tp) * 8); }

iterator insert(const _Tp &start, const _Tp &end) {
    typename std::list< std::pair<_Tp, _Tp> >::iterator iter, next, stopper;

    assert(start <= end);

    for (iter = _range.begin(), stopper = _range.end(); iter != stopper; ++iter) {
        if (end < iter->first) { // insert new first entry, or extend current first entry
            if ((end + 1) == iter->first) {
                iter->first = start;

                return iterator(iter, stopper, 0);
            } else {
                return iterator(_range.insert(iter, std::pair<_Tp, _Tp>(start, end)), stopper, 0);
            }
        } else if ((start - 1) <= iter->second) { // fold into successive entries
            if (end <= iter->second) {
                return iterator(iter, stopper, start - iter->first);
            } else {
                next = iter;
                ++next;

                while ((next != stopper) && (next->second < end)) {
                    next = _range.erase(next);
                }

                if (next == stopper) {
                    iter->second = end;
                    return iterator(iter, stopper, start - iter->first);
                } else {
                    if (next->first - 1 <= start) {
                        iter->second = next->second;

                        _range.erase(next);
                    } else {
                        iter->second = end;
                    }

                    return iterator(iter, stopper, start - iter->first);
                }
            }
        }
    }

    return iterator(_range.insert(stopper, std::pair<_Tp, _Tp>(start, end)), stopper, 0);
}

iterator insert(const _Tp &__x) {
    return insert(__x, __x);

    /*std::list< std::pair<_Tp, _Tp> >::iterator previous, iter, stopper;

    for (iter = _range.begin(), stopper = previous = _range.end(); iter != stopper; ++iter) {
        if (iter->first > __x) {
            if (iter->first - 1 == __x) {
                if (previous != stopper) {
                    if (previous->second + 1 == __x) { // Merge existing entries
                        previous->second = iter->second;

                        _range.erase(iter);

                        return iterator(previous, _range.end(), __x - previous->first);
                    } else {
                        --(iter->first);
                        return iterator(iter, stopper, 0);
                    }
                }
            } else {
                return iterator(_range.insert(iter, std::pair<_Tp, _Tp>(__x, __x)), stopper, 0);
```

```
                }
            } else if (iter->second >= __x) {
                return iterator(iter, stopper, __x - iter->first);
            }

            previous = iter;
        }

        return iterator(_range.insert(_range.end(), std::pair<_Tp, _Tp>(__x, __x)));*/
    }

    void clear(void) { _range.clear(); }

    iterator remove(const _Tp &start, const _Tp &end) {
        typename std::list< std::pair<_Tp, _Tp> >::iterator iter, stopper;

        assert(start <= end);

        for (iter = _range.begin(), stopper = _range.end(); iter != stopper; /* Do nothing */) {
            if (end < iter->first) {
                return iterator(iter, stopper, 0);
            } else {
                if (start <= iter->first) { // Erase from start
                    if (end >= iter->second) { // Erase all
                        iter = _range.erase(iter);
                    } else { // Erase some
                        iter->first = end + 1;
                        return iterator(iter, stopper, 0);
                    }
                } else if (end >= iter->second) { // Erase from end (which cannot be all the way to start
                    if (start <= iter->second) { // Erase some
                        iter->second = start - 1;
                    } // else this one doesn't apply to use

                    ++iter;
                } else { // Erase subset
                    _range.insert(iter, std::pair<_Tp, _Tp>(iter->first, start - 1));

                    iter->first = end + 1;

                    return iterator(iter, stopper, 0);
                }
            }
        }

        return iterator(stopper, stopper, 0);
    }

    iterator remove(const _Tp &__x) {
        return remove(__x, __x);
    }

    iterator erase(iterator __position) {
        erase(*__position);
    }

    iterator erase(iterator __first, iterator __last) {
        erase(*__first, *__last);
    }

    void swap(range& __x) {
        _range.swap(__x.range);
    }
};


template<typename _Tp> int readRange(range<_Tp> &r, const char *input) {
    char *curPos = const_cast<char*>(input); // Yep, it's naughty, but strtoll does it, and forces us to do it too.
    _Tp firstValueInRange;
    long long rawValue;
    bool amReadingRange = false;
    int err = 0;

    do {
        rawValue = strtoll(curPos, &curPos, 0);

        if (input == curPos) { // Handle the case where we can't read anything at all
#ifndef NDEBUG
            fprintf(stderr, "%s %s:%d - Unable to interpret \"%s\".\n", __FILE__, __func__, __LINE__, curPos);
#endif
            err = EINVAL;
        } else {
            while (isspace(*curPos)) {
                ++curPos;
            }

            if ('-' == *curPos) {
                ++curPos;

                if (amReadingRange) {
#ifndef NDEBUG
                    fprintf(stderr, "%s %s:%d - Already reading a range and encountered another hyphen %lu bytes in (at \"%s\").\n", __FILE__, __func__, __LINE__,
(intptr_t)curPos - (intptr_t)input, curPos);
#endif
                    err = EINVAL;
                } else {
                    amReadingRange = true;

                    firstValueInRange = static_cast<_Tp>(rawValue);
                }
            } else {
                while (',' == *curPos) {
                    ++curPos;
                }

                if (amReadingRange) {
```

```
                    r.insert(firstValueInRange, static_cast<_Tp>(rawValue));

                    amReadingRange = false;
                } else {
                    r.insert(static_cast<_Tp>(rawValue));
                }
            }
        }
    } while ((0 == err) && (0 != *curPos));

    return err;
}

template<typename _Tp> std::ostream& operator<<(std::ostream& stream, const range<_Tp> &r) {
    typename std::list< std::pair<_Tp, _Tp> >::const_iterator iter, stopper;
    bool havePrintedFirstValue = false;

    for (iter = r._range.begin(), stopper = r._range.end(); iter != stopper; ++iter) {
        if (havePrintedFirstValue) {
            stream << ',';
        }

        stream << iter->first;

        if (iter->first != iter->second) {
            stream << '-' << iter->second;
        }

        havePrintedFirstValue = true;
    }

    return stream;
}
```