Recruiting test

Test assignment

Thank you for participating in our recruiting test. This will be a C++ programming test!

How to prepare for this test

Task Description

`interval_map<K,V>` is a data structure that associates keys of type *K* with values of type *V*. It is designed to be used efficiently in situations where intervals of consecutive keys are associated with the same value. Your task is to implement the `assign` member function of this data structure, which is outlined below.

`interval_map<K, V>` is implemented on top of `std::map`. For more information on `std::map`, you may refer to cppreference.com.

Each key-value-pair (*k*,*v*) in `interval_map<K,V>::m_map` means that the value *v* is associated with all keys from *k* (including) to the next key (excluding) in `m_map`. The member `interval_map<K,V>::m_valBegin` holds the value that is associated with all keys less than the first key in `m_map`.

Example: Let *M* be an instance of `interval_map<int,char>` where

```
M.m_valBegin=='A',
M.m_map=={ (1,'B'), (3,'A') },
```

Then *M* represents the mapping

```
...
-2 -> 'A'
-1 -> 'A'
0 -> 'A'
1 -> 'B'
2 -> 'B'
3 -> 'A'
```

`4 -> 'A'`
`5 -> 'A'`
...

The representation in the `std::map` must be canonical, that is, consecutive map entries must not contain the same value: `..., (3,'A'), (5,'A'), ...` is not allowed. Likewise, the first entry in `m_map` must not contain the same value as `m_valBegin`. Initially, the whole range of *K* is associated with a given initial value, passed to the constructor of the `interval_map<K,V>` data structure.

Key type *K*

- besides being copyable and assignable, is less-than comparable via `operator<`, and
- does not implement any other operations, in particular no equality comparison or arithmetic operators.

Value type *V*

- besides being copyable and assignable, is equality-comparable via `operator==`, and
- does not implement any other operations.

First Evaluation

```
#include <map>
template<typename K, typename V>
class interval_map {
        friend void IntervalMapTest();
        V m_valBegin;
        std::map<K,V> m_map;
public:
        // constructor associates whole range of K with val
        interval_map(V const& val)
        : m_valBegin(val)
        {}

        // Assign value val to interval [keyBegin, keyEnd).
        // Overwrite previous values in this interval.
        // Conforming to the C++ Standard Library conventions, the interval
        // includes keyBegin, but excludes keyEnd.
        // If !( keyBegin < keyEnd ), this designates an empty interval,
```

```
        // and assign must do nothing.
        void assign( K const& keyBegin, K const& keyEnd, V const& val ) {


// INSERT YOUR SOLUTION HERE

                // First check to see if any operation needs to be done
                // at all and if not return
                //
                if(!(keyBegin<keyEnd)){

                        // exit the function
                        //
                        return;
                }

        // get the value that currently takes up the first index of
                // space you want to fill
        //
        auto IBegin = m_map.lower_bound(keyBegin);

        // get the next value to see if you would overlap by inserting
                // the interval given
        //
        auto IEnd = m_map.lower_bound(keyEnd);

                /* check to see if there is overlap or if you've reached the en
                /*Referenced later as overlapped cases*/

                // declaration of pair that may or may not be used
                //
                std::pair<K,V> mypair;

                // Flag in here to see if pair has been changed
                //
        mypair.first = (V) NULL;

                // First condition checks to see if our interval end is at the
                // of the map
                // Second condition check to see if we the end of our interval
                // overlap the next interval
                //
        if((IEnd == m_map.end()) || (keyEnd < IEnd->first)){
```

```
                // If either condition is met, we need to update mypair
                // be inserted at the end so we don't lose the extra in
                //

                // Resumes the pair at the end of our inserted pair
                //
    mypair.first = keyEnd;

                // Uses the given function to give the pair the old val
                //
    mypair.second = operator[](keyEnd);
}

        // First condition checks to see if our interval begin is at th
        // of the map
        // Second condition checks to see if we are trying to insert so
        // the middle of a value
        //
else if ((IBegin == m_map.end()) || (keyBegin < IBegin->first)){

                // If either condition is met, we need to update mypair
                // be inserted at the end so we don't lose the extra in
                //

                // Resumes the pair at the end of our inserted pair
                //
                mypair.first = keyEnd;

                // Uses the given function to give the pair the old val
                //
    mypair.second = operator[](keyBegin);
}

        // if not inserted at end erase all the inbetween pairs
        //
        if(IEnd != m_map.end()){
        m_map.erase(IBegin,IEnd);
}

        // create a pair to insert change
        //
        std::pair<K,V>insertpair;
```

```
        insertpair.first = keyBegin;
        insertpair.second = val;

                // insert change
                //
        m_map.insert(insertpair);

                // check to see if overlapped case was generated
                //
        if(mypair.first!=(V) NULL){

                        // if so insert new pair
                        //
                        m_map.insert(mypair);
                }




        }

        // look-up of the value associated with key
        V const& operator[]( K const& key ) const {
                auto it=m_map.upper_bound(key);
                if(it==m_map.begin()) {
                        return m_valBegin;
                } else {
                        return (--it)->second;
                }
        }
};

// Many solutions we receive are incorrect. Consider using a randomized test
// to discover the cases that your implementation does not handle correctly.
// We recommend to implement a test function that tests the functionality of
// the interval_map, for example using a map of int intervals to char.
```

Unfortunately, this program failed to meet the criterion marked in red:

Type requirements are met: You must adhere to the specification of the key and value type given above.

- **Correctness:** Your program should produce a working `interval_map` with the behavior described above. In particular, pay attention to the validity of iterators. It is illegal to dereference end iterators. Consider using a checking STL implementation such as the one shipped with Visual C++ or GCC.
- **Canonicity:** The representation in `m_map` must be canonical.
- **Running time:** Imagine your implementation is part of a library, so it should be big-O optimal. In addition:
  - Do not make big-O more operations on *K* and *V* than necessary because you do not know how fast operations on *K*/*V* are; remember that constructions, destructions and assignments are operations as well.
  - Do not make more than one operation of amortized O(log *N*), in contrast to O(1), running time, where *N* is the number of elements in `m_map`.
  - Otherwise favor simplicity over minor speed improvements.

Second Chance

Your first try failed. But we give you one more chance to fix your code:

Final Evaluation

```
#include <map>
template<typename K, typename V>
class interval_map {
        friend void IntervalMapTest();
        V m_valBegin;
        std::map<K,V> m_map;
public:
        // constructor associates whole range of K with val
        interval_map(V const& val)
        : m_valBegin(val)
        {}

        // Assign value val to interval [keyBegin, keyEnd).
        // Overwrite previous values in this interval.
        // Conforming to the C++ Standard Library conventions, the interval
        // includes keyBegin, but excludes keyEnd.
```

```
        // If !( keyBegin < keyEnd ), this designates an empty interval,
        // and assign must do nothing.
        void assign( K const& keyBegin, K const& keyEnd, V const& val ) {


// INSERT YOUR SOLUTION HERE

                // First check to see if any operation needs to be done
                // at all and if not return
                //
                if(!(keyBegin<keyEnd)){

                        // exit the function
                        //
                        return;
                }

        // get the value that currently takes up the first index of
                // space you want to fill
        //
        auto IBegin = m_map.lower_bound(keyBegin);

        // get the next value to see if you would overlap by inserting
                // the interval given
        //
        auto IEnd = m_map.lower_bound(keyEnd);

                /* check to see if there is overlap or if you've reached the en
                /*Referenced later as overlapped cases*/

                // declaration of pair that may or may not be used
                //
                std::pair<K,V> mypair;

                // Flag in here to see if pair has been changed
                //
        bool extrapair = false;

                // First condition checks to see if our interval end is at the
                // of the map
                // Second condition check to see if we the end of our interval
                // overlap the next interval
                //
```

```
if((IEnd == m_map.end()) || (keyEnd < IEnd->first)){

                // If either condition is met, we need to update mypair
                // be inserted at the end so we don't lose the extra in
                //

                // Resumes the pair at the end of our inserted pair
                //
    mypair.first = keyEnd;

                // Uses the given function to give the pair the old val
                //
    mypair.second = operator[](keyEnd);

                // set the flag to have a pair change
                //
                extrapair = true;
}

        // First condition checks to see if our interval begin is at th
        // of the map
        // Second condition checks to see if we are trying to insert so
        // the middle of a value
        //
else if ((IBegin == m_map.end()) || (keyBegin < IBegin->first)){

                // If either condition is met, we need to update mypair
                // be inserted at the end so we don't lose the extra in
                //

                // Resumes the pair at the end of our inserted pair
                //
                mypair.first = keyEnd;

                // Uses the given function to give the pair the old val
                //
    mypair.second = operator[](keyBegin);

                // set the flag to have a pair change
                //
                extrapair = true;
}
```

```
                // if not inserted at end erase all the inbetween pairs
                //
                if(IEnd != m_map.end()){
                m_map.erase(IBegin,IEnd);
        }

                // create a pair to insert change
                //
                std::pair<K,V>insertpair;
        insertpair.first = keyBegin;
        insertpair.second = val;

                // insert change
                //
        m_map.insert(insertpair);

                // check to see if overlapped case was generated
                //
        if(extrapair == true){

                        // if so insert new pair
                        //
                        m_map.insert(mypair);
                }




        }

        // look-up of the value associated with key
        V const& operator[]( K const& key ) const {
                auto it=m_map.upper_bound(key);
                if(it==m_map.begin()) {
                        return m_valBegin;
                } else {
                        return (--it)->second;
                }
        }
};

// Many solutions we receive are incorrect. Consider using a randomized test
```

```
// to discover the cases that your implementation does not handle correctly.
// We recommend to implement a test function that tests the functionality of
// the interval_map, for example using a map of int intervals to char.
```

Unfortunately, this program also did not pass.

This time, it failed to meet this criterion:

- **Type requirements are met:** You must adhere to the specification of the key and value type given above. For example, many solutions we receive use operations other than those that are explicitly stated in the task description. We have to reject many solutions because they assume that *V* is default-constructible, e.g., by using std::map::operator[].

- **Correctness:** Your program should produce a working `interval_map` with the behavior described above. In particular, pay attention to the validity of iterators. It is illegal to dereference end iterators. Consider using a checking STL implementation such as the one shipped with Visual C++ or GCC. Many solutions we receive do not create the data structure that was asked for, e.g., some interval ends up being associated with the wrong value. Others contain a code path that will eventually dereference an invalid or end iterator.
- **Canonicity:** The representation in `m_map` must be canonical. Some solutions we receive create consecutive map entries containing the same value, or repeat the value of m_valBegin at the beginning of the map.
- **Running time:** Imagine your implementation is part of a library, so it should be big-O optimal. In addition:
    - Do not make big-O more operations on *K* and *V* than necessary because you do not know how fast operations on *K*/*V* are; remember that constructions, destructions and assignments are operations as well.
    - Do not make more than one operation of amortized O(log *N*), in contrast to O(1), running time, where *N* is the number of elements in `m_map`.
    - Otherwise favor simplicity over minor speed improvements.

We regret that we cannot provide you with information specific to your solution, or with a correct version of the algorithm, because if we did, then we could no longer use this challenge for our interview process. We sincerely hope for your understanding on this matter.

Since this was your final submission, we have decided not to offer you an interview.

We want to thank you for your interest in the C++ developer position at think-cell and for the time and effort you have put into taking our programing test. We know that C++ developers are a scarce resource and that you have a choice of companies you can work for. For this reason we highly value your application.

Arno, our CTO, and the HR Team want to thank you for your time and interest in our company, and wish you the very best in your future career.