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 efficiently associates intervals of keys of type K with values of type V. 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. In case you are not entirely sure which functions std::map provides, what they do and which guarantees they provide, we provide an excerpt of the C++ standard here: More

Each key-value-pair (k,v) in the std::map means that the value v is associated with the interval from k (including) to the next key (excluding) in the std::map.

Example: the std::map (0,'A'), (3,'B'), (5,'A') represents the mapping

0 -> 'A'
1 -> 'A'
2 -> 'A'
3 -> 'B'
4 -> 'B'
5 -> 'A'
6 -> 'A'
7 -> 'A'
... all the way to numeric_limits<int>::max()
The representation in the std::map must be canonical, that is, consecutive map entries must not have the same value: ..., (0,'A'), (3,'A'), ... is not allowed. 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<
- is bounded below, with the lowest value being std::numeric_limits<K>::lowest()
- 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==
- does not implement any other operations

First Evaluation

```
#include <map>
#include <limits>

template<typename K, typename V>
class interval_map {
        std::map<K,V> m_map;

public:
    // constructor associates whole range of K with val by inserting (K_mir
    // into the map
    interval_map( V const& val) {
        m_map.insert(m_map.end(),std::make_pair(std::numeric_limits<K>::low
    }

    // 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 ) {
```

```
// Empty interval, do nothing.
if (!(keyBegin < keyEnd))
{
   return;
}

// Find key such that key >= keyBegin or map.end(), first O(log N) op
auto lbKeyBegin = m_map.lower_bound(keyBegin);

// Find key such that keyEnd < key, or map.end(), second O(log N) ope
auto ubKeyEnd = m_map.upper_bound(keyEnd);

// The entry just before the end element in the range
auto prevUBKeyEnd = std::prev(ubKeyEnd);

// The new end iterator based on the input val, it could be map.end()
typename std::map<K, V>::iterator newEnd;

if (prevUBKeyEnd->second == val)
{
   // Value being inserted is same as the previous value
   newEnd = ubKeyEnd;
}
else
{
   // Values are not same, check the keys
   if (!(prevUBKeyEnd->first < keyEnd) && !(keyEnd < prevUBKeyEnd->fi
   {
      // prevUBKeyEnd is same
      newEnd = prevUBKeyEnd;
   }
   else
   {
      // prevUBKeyEnd is less than the new end being inserted
      newEnd = m_map.insert_or_assign(ubKeyEnd, keyEnd, prevUBKeyEnd-
   }
}

// The new begin iterator based on the input val, it could be map.beg
typename std::map<K, V>::iterator newBegin;
```

```
    if (lbKeyBegin != m_map.begin())
    {
       auto prevLBKeyBegin = std::prev(lbKeyBegin);
       if (!(prevLBKeyBegin->second == val))
       {
          // New value is not same as the existing value.
          newBegin = m_map.insert_or_assign(lbKeyBegin, keyBegin, val);
       }
       else
       {

          // New value is same as existing value. Merge newBegin with pre
          newBegin = prevLBKeyBegin;
       }
    }
    else
    {
       // Previous interval does not exist
       newBegin = m_map.insert_or_assign(lbKeyBegin, keyBegin, val);
    }

    // Erase all map elements between new begin and end, so that we have
    auto nextBegin = std::next(newBegin);
    if (nextBegin != m_map.end())
    {
       // Complexity : std::distance(nextBegin, newEnd)
       while (nextBegin != newEnd)
       {
          auto toErase = nextBegin;
          ++nextBegin;
          m_map.erase(toErase);
       }
    }
```

```
}

// look-up of the value associated with key
V const& operator[]( K const& key ) const {
    return ( --m_map.upper_bound(key) )->second;
```

```
    }
};

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

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 two operations of amortized O(log N), in contrast to O(1), running time, where N is the number of elements in m_map. Any operation that needs to find a position in the map "from scratch", without being given a nearby position, is such an operation.
  - 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>
```

```
#include <limits>

template<typename K, typename V>
class interval_map {
        std::map<K,V> m_map;

public:
    // constructor associates whole range of K with val by inserting (K_mir
    // into the map
    interval_map( V const& val) {
        m_map.insert(m_map.end(),std::make_pair(std::numeric_limits<K>::low
    }


    // 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 ) {
```

```
      // Empty interval, do nothing.
      if (!(keyBegin < keyEnd))
      {
         return;
      }

      // Find key such that key >= keyBegin or map.end(), first O(log N) op
      auto lbKeyBegin = m_map.lower_bound(keyBegin);

      // Find key such that keyEnd < key, or map.end(), second O(log N) ope
      auto ubKeyEnd = m_map.upper_bound(keyEnd);

      // The entry just before the end element in the range
      auto prevUBKeyEnd = std::prev(ubKeyEnd);

      // The new end iterator based on the input val, it could be map.end()
      typename std::map<K, V>::iterator newEnd;
```

```
        if (prevUBKeyEnd->second == val)
        {
           // Value being inserted is same as the previous value
           newEnd = ubKeyEnd;
        }
        else
        {
           // Values are not same, check the keys
           if (!(prevUBKeyEnd->first < keyEnd) && !(keyEnd < prevUBKeyEnd->fi
           {
              // prevUBKeyEnd is same
              newEnd = prevUBKeyEnd;
           }
           else
           {
              // prevUBKeyEnd is less than the new end being inserted
              newEnd = m_map.insert_or_assign(ubKeyEnd, keyEnd, prevUBKeyEnd-
           }
        }

        // The new begin iterator based on the input val, it could be map.beg
        typename std::map<K, V>::iterator newBegin;
        if (lbKeyBegin != m_map.begin())
        {
           auto prevLBKeyBegin = std::prev(lbKeyBegin);
           if (!(prevLBKeyBegin->second == val))
           {
              // New value is not same as the existing value.

              newBegin = m_map.insert_or_assign(lbKeyBegin, keyBegin, val);
           }
           else
           {
              // New value is same as existing value. Merge newBegin with pre
              newBegin = prevLBKeyBegin;
           }
        }
        else
        {
           // Previous interval does not exist
           newBegin = m_map.insert_or_assign(lbKeyBegin, keyBegin, val);
```

```
        }

        // Erase all map elements between new begin and end, so that we have
        auto nextBegin = std::next(newBegin);
        if (nextBegin != m_map.end())
        {
            // Linear complexity
            m_map.erase(nextBegin, newEnd);
        }
```

```
    }

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

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

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 have consecutive map entries containing the same value.
- **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 two operations of amortized O(log N), in contrast to O(1), running time, where N is the number of elements in m_map. Any operation that needs to find a position in the map "from scratch", without being given a nearby position, is such an operation.
  - 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.