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

Install an editor and C++ compiler of your choice. Your solution will be compiled with GCC 7.2.0 (with -std=c++17), and you will have the chance to fix any compilation errors prior to submission. You may use the standard libraries for the task. It is not permitted to use other libraries, however you will not need them anyway.

You are free in your choice of operating system and development environment. The task is a very general programming assignment testing general problem structuring and programming proficiency. The solution has to be submitted within a 9 hour time frame. Excellent C++ specialists will of course solve the problem in **3 to 4 hours**, and we leave it to the candidates to test their solutions thoroughly before handing them in (in case they finish early).

You must develop the solution yourself. You may not let others help you or search for existing solutions. Of course you may use any documentation of the C++ language or the C++ Standard Library. Do not give your solution to others or make it public. It will entice others into sending in plagiarized solutions. If you use an online compiler, make sure that the privacy settings are set to private. Publishing the task description is also considered cheating and will void your application.

The amount of code you need for a correct solution is relatively small, and you will spend most of the allocated time thinking rather than typing. We would like to receive the best solution you can come up with!

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: Less

The following paragraphs from the final draft of the C++1x ISO standard describe the avai operations on a std::map container, their effects and their complexity.

23.2.1 General container requirements

§1 Containers are objects that store other objects. They control allocation and deallocation these objects through constructors, destructors, insert and erase operations.

§6 begin() returns an iterator referring to the first element in the container. end() returns an iterator which is the past-the-end value for the container. If the container is empty, then begin() == end();

24.2.1 General Iterator Requirements

§1 Iterators are a generalization of pointers that allow a C++ program to work with differe data structures.

§2 Since iterators are an abstraction of pointers, their semantics is a generalization of mos of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers.

§5 Just as a regular pointer to an array guarantees that there is a pointer value pointing pa the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called past-the-end v Values of an iterator i for which the expression *i is defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable. Iterators can also singular values that are not associated with any sequence. [Example: After the declaration an uninitialized pointer x (as with int* x;), x must always be assumed to have a singular value of a pointer. -end example] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that satisfy the DefaultConstructible requirements, using a value-initialized iterator as the source of a copy or move operation.

§10 An invalid iterator is an iterator that may be singular. (This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.)

23.2.4 Associative containers

§1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

§4 An associative container supports unique keys if it may contain at most one element fo key. Otherwise, it supports equivalent keys. The set and map classes support unique keys; multiset and multimap classes support equivalent keys.

§5 For map and multimap the value type is equal to std::pair<const Key, T>. Keys in an associative container are immutable.

§6 iterator of an associative container is of the bidirectional iterator category.
(i.e., an iterator i can be incremented and decremented: ++i; --i;)

§9 The insert member functions (see below) shall not affect the validity of iterators and references to the container, and the erase members shall invalidate only iterators and references to the erased elements.

§10 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defi the comparison that was used to construct them.

Associative container requirements (in addition to general container requirements):

std::pair<iterator, bool> insert(std::pair<const key_type, T> const'' t)
Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equival to the key of t.
Complexity: logarithmic

iterator insert(const_iterator p, std::pair<const key_type, T> const'' t)
Effects: Inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys. Always returns the iterator pointing to the element with key equivalent to the key of t.
Complexity: logarithmic in general, but amortized constant if t is inserted right before p.

size_type erase(key_type const'' k)
Effects: Erases all elements in the container with key equivalent to k. Returns the number erased elements.
Complexity: log(size of container) + number of elements with key k

iterator erase(const_iterator q)
Effects: Erases the element pointed to by q. Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, retu end().

Complexity: Amortized constant

iterator erase(const_iterator q1, const_iterator q2)
Effects: Erases all the elements in the left-inclusive and right-exclusive range [q1,q2).
Returns q2.
Complexity: Amortized O(N) where N has the value distance(q1, q2).

void clear()
Effects: erase(begin(), end())
Post-Condition: empty() returns true
Complexity: linear in size().

iterator find(key_type const" k);
Effects: Returns an iterator pointing to an element with the key equivalent to k, or end() if such an element is not found.
Complexity: logarithmic

size_type count(key_type constquot;& k)
Effects: Returns the number of elements with key equivalent to k
Complexity: log(size of map) + number of elements with key equivalent to k

iterator lower_bound(key_type const" k)
Effects: Returns an iterator pointing to the first element with key not less than k, or end() if such an element is not found.
Complexity: logarithmic

iterator upper_bound(key_type const" k)
Effects: Returns an iterator pointing to the first element with key greater than k, or end() if such an element is not found.
Complexity: logarithmic

23.4.1 Class template map

§1 A map is an associative container that supports unique keys (contains at most one of e key value) and provides for fast retrieval of values of another type T based on the keys. Th map class supports bidirectional iterators.

23.4.1.2 map element access

T" operator[](const key_type" x);
Effects: If there is no key equivalent to x in the map, inserts value_type(x, T()) into the ma
Returns: A reference to the mapped_type corresponding to x in *this.
Complexity: logarithmic.

T" at(const key_type" x);
const T" at(const key_type" x) const;
Returns: A reference to the element whose key is equivalent to x.

Throws: An exception object of type out_of_range if no such element is present.
Complexity: logarithmic.

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

You are given the following source code:

```
#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_min, val)
    // into the map
    interval_map( V const& val) {
        m_map.insert(m_map.end(),std::make_pair(std::numeric_limits<K>::lowest(),v
    }

    // 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 ) {
```

```
                                        --beforeMid;
                                        if (beforeMid->second == val)
                                                insertMid = false;
                                }
                        }



                        if (endHasExtra) {
                                if ( (insertMid && endExtra.second == val) ||
(!insertMid && endExtra.second == beginExtra.second) )
                                        endHasExtra = false;
                        } else {
                                if ( (insertMid && itEnd!=m_map.end() && itEnd-
>second == val) || (!insertMid && itEnd!=m_map.end() && itEnd->second ==
beginExtra.second) )
                                        itEnd = m_map.erase(itEnd);
                        }

                        itBegin = m_map.erase(itBegin, itEnd);
                        if (beginHasExtra)
```

```
    }

    // 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 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 unsigned int intervals to char.
```

You can download this source code here: Download

Your task is to implement the function `assign`. Your implementation is graded by these criteria in this order:

- **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.

You should not take longer than 9 hours, but you may of course be faster. Do not rush, we would not give you this assignment if it were trivial.

When you are done, please complete the form and click Compile. You can improve and compile solutions as often as you like.
Please submit your solution until 10:53 UTC.

Compile

Further instructions will be given once your code compiles correctly.