**Elements That Must be Implemented**

1. **assign** member function, which efficiently associates intervals of keys of type K with values of type V
2. Key type K: less-than comparable via **operator<**
3. Value type V: equality-comparable via **operator==**

**Assign Member Function Behavior**

1. **Initially, the whole range of K is associated with a given initial value, passed to the constructor**.
2. Each key-value-pair (k,v) in the m\_map member means that the **value v is associated to the interval from k (including) to the next key (excluding) in m\_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<key>::max()**

**Restrictions**

1. Key type K

* 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**

1. Value type V

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

1. The representation **in m\_map must be canonical**, that is, **consecutive map entries must not have the same value**: ..., (**0,'A'), (3,'A'), ... is not allowed**.
2. Do not make big-O more operations on K and V than necessary
3. **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

**Evaluation Criteria**

1. **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++.**
2. **Use functions of std::map wherever you can**
3. 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**.
4. **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.
5. Otherwise favor simplicity over minor speed improvements.

**Challenges Faced**

1. The constructor that is given with the assignment code does not do what the assignment text claims, which led to some confusion. This means that the initial value needs to be stored so that it can be set dynamically for key range gaps. Implementing the algorithm of key-range gap dynamic initialValue filling, which ensures that the map is filled with the initialValue only when needed, will save memory and execution time
2. The restrictions that were imposed in the assignment. Couldn’t see the value or need for implementing the requested operators. The assignment author wants the assignment to be done in a certain way, giving clues perhaps, which led to more thinking on my part. The assignment in itself and getting the end result done is not difficult, despite myself not having any prior experience with std::map. What was difficult was trying to get to the solution that the assignment author specifically wants while following all the given clues
3. Some inconsistencies, such as the constructor, and not being able to ever change the very last value in the map, made me some extra time thinking about the objective and if there were any errors in the assignment text
4. Although I could see the need for implementing an operator< on key after running the tests due to unsigned int variable overflow when testing with negative numbers; I would not be able to implement an operator< override form within the assign function, since that is against the c++ language rules. I have to implement it in the class. In either case, I wouldn’t be able to submit that code, since you only take the assign function body in the online form. This was quite confusing! I spent some time trying to think about this and figure it out. At the end I decided to implement a generic check for the key type overflow

**Original Assignment Specification**

1. 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
2. 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++1x draft standard below
3. Each key-value-pair (k,v) in the m\_map member means that the **value v is associated to the interval from k (including) to the next key (excluding) in m\_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<key>::max()**

1. The **representation in m\_map must be canonical**, that is, **consecutive map entries must not have the same value**: ..., (**0,'A'), (3,'A'), ... is not allowed**.
2. **Initially, the whole range of K is associated with a given initial value, passed to the constructor**.
3. 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**

1. Value type V

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

Your task is to implement the function "assign".

**Your implementation is graded by these criteria in this order:**

1. Correctness (of course): 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++.**
2. Simplicity: Simple code is easy to understand and maintain, which is important in large projects. To write a simple solution, you need to exploit the structure of the problem. **Use functions of std::map wherever you can**.
3. 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.

1. Time to turn in the solution: 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.

**std::map description**

The following paragraphs from the final draft of the C++1x ISO standard describe the available

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 of

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 different

data structures.

§2 Since iterators are an abstraction of pointers, their semantics is a generalization of most

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 past

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 values.

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 have

singular values that are not associated with any sequence. [ Example: After the declaration of

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 for each

key. Otherwise, it supports equivalent keys. The set and map classes support unique keys; the

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 defined by

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 equivalent

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 of

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, returns

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 each

key value) and provides for fast retrieval of values of another type T based on the keys. The

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 map.

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.