

Standard Template Library 3

Marco Lattuada, Mehrnoosh Askarpour,
Danilo Ardagna

Politecnico di Milano

name.lastname@polimi.it

REVISITING ITERATORS

Revisiting Iterators

- The most fundamental property of any algorithm is the list of operations it requires from its iterator(s)
- Some algorithms, such as `find`, require only the ability to access an element through the iterator, to increment the iterator, and to compare two iterators for equality
- Others, such as `sort`, require the ability to read, write, and randomly access elements

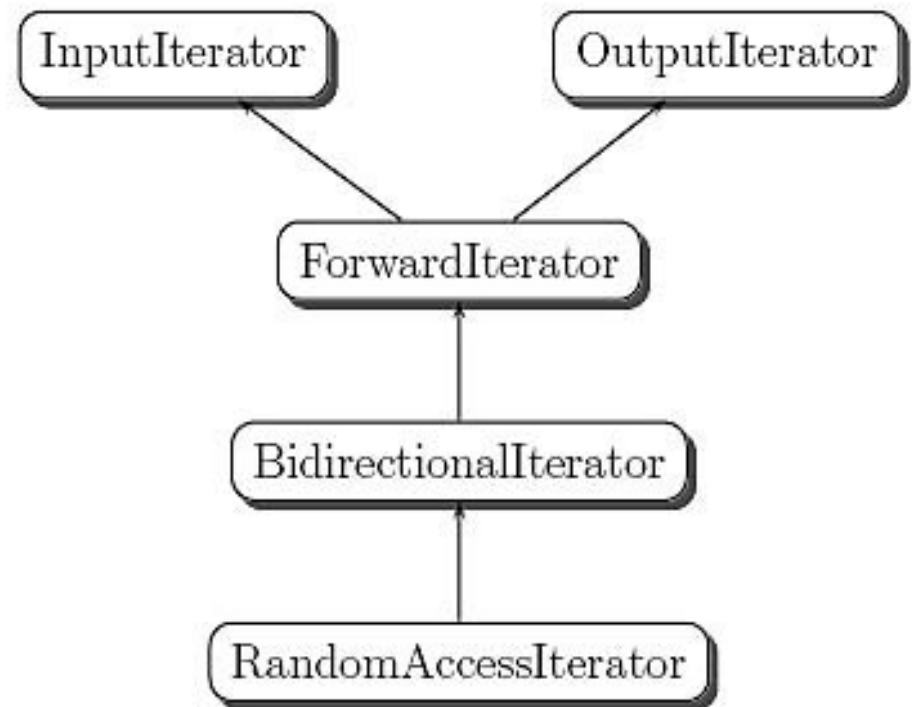
The five iterator categories

The **iterator operations** required by the algorithms are grouped into five iterator categories.

Iterators are categorized by the operations they provide and the categories form a sort of hierarchy.

An iterator of a higher category provides all the operations of the iterators of a lower category.

For each parameter, the iterator must be at least as powerful as the stipulated minimum. Passing an required iterator. Use a less powerful iterator gives a Compilation error.



The five iterator categories

	<code>==, !=</code>	<code>++</code>	<code>*, -></code>	Multi-pass	<code>--</code>	<code><, <=, >, >=</code>	<code>+, +=, -, -=</code>	<code>[]</code>
Input iterators	X	X	X					
Output iterators		X	X					
Forward iterators	X	X	X	X				
Bidirectional iterators	X	X	X	X	X			
Random access iterators	X	X	X	X	X	X	X	X

Input iterators

- Read, **single-pass**, increment
- Equality and inequality operators (`==`, `!=`) to compare two iterators
- Prefix and postfix increment (`++`) to advance the iterator
- Dereference operator (`*`) to read an element; dereference may appear **only on the right-hand side of an assignment**
- The arrow operator (`->`) as a synonym for `(* it).member`—that is, dereference the iterator and fetch a member from the underlying object
- The ***find*** and ***accumulate*** algorithms require input iterators

Reading Algorithms: accumulate

```
double total = 0.0;
for (multiset<int>::iterator itr =
values.begin(); itr != values.end(); ++itr)
    total += *itr;
cout << "Average is: " << total /
values.size() << endl;
```



```
cout << accumulate(values.begin(),
values.end(), 0.0) / values.size() << endl;
```

Output iterators

- Write, single-pass, increment
- Prefix and postfix increment (++) to advance the iterator
- Dereference (*), which may appear only as the **left-hand side of an assignment** (assigning to a dereferenced output iterator writes to the underlying element)
- The **copy** algorithm uses an output iterator

Forward iterators

- Read/Write, multi-pass, increment
- They move in only one direction through the sequence
- They can read or write the same element multiple times
- The *replace* algorithm requires a forward iterator
- Iterators on `forward_list` are forward iterators

Bidirectional iterators

- Read/Write, multi-pass, increment/decrement
- The `reverse` algorithm requires bidirectional iterators
- Aside from `forward_list`, the library containers supply iterators that meet the requirements for a bidirectional iterator

Random access Iterators

- Read/Write, multi-pass, full arithmetic
- The relational operators ($<$, $<=$, $>$, and $>=$) to compare the relative positions of two iterators
- Addition and subtraction operators ($+$, $+=$, $-$, and $-=$) on an iterator and an integral value. The result is the iterator advanced (or retreated) the integral number of elements within the sequence
- The subtraction operator ($-$) when applied to two iterators, yields the distance between two iterators
- The subscript operator ($\text{iter}[n]$) as a synonym for $*$ ($\text{iter} + n$)
- The *sort* algorithms require random-access iterators
- Iterators for *array*, *deque*, *string*, and *vector* are random-access iterators

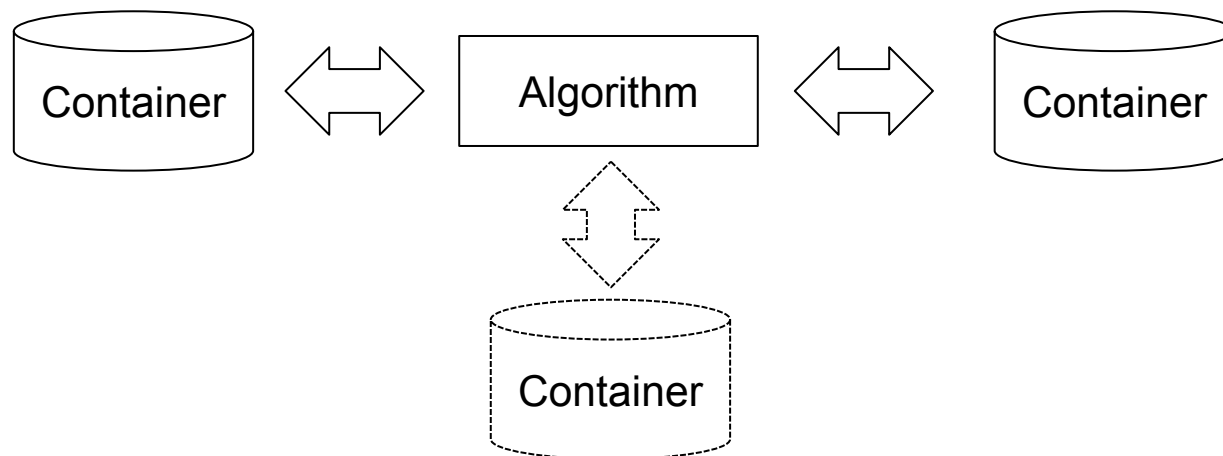
Algorithms

STL Generic Algorithms

- Rather than defining operations as members of each container type, the standard library defines a **set of generic algorithms** that operate on elements of differing type and across multiple container types
- In general, the **algorithms** do not work directly on a container. Instead, they **operate by traversing a range of elements** bounded by two iterators

STL Generic Algorithms

- The algorithms library defines functions for a **variety of purposes** (e.g. searching, sorting, counting, manipulating) that operate on ranges [first, last) of elements
- The **data** and **operations** in STL are **decoupled**. Container classes manage the data, and the operations are defined by the algorithms, used together with the **iterators**



Some useful standard algorithms

- `r=find(b, e, v)` `r` points to the first occurrence of `v` in `[b,e)`
- `r=find_if(b, e, p)` `r` points to the first element `x` in `[b,e)` for which `p(x)`
- `x=count(b, e, v)` `x` is the number of occurrences of `v` in `[b,e)`
- `x=count_if(b, e, p)` `x` is the number of elements in `[b,e)` for which `p(x)`
- `sort(b, e)` sort `[b,e)` using `<`
- `sort(b, e, p)` sort `[b,e)` using `p`
- `copy(b, e, b2)` copy `[b,e)` to `[b2, b2+(e-b))`
there had better be enough space after `b2`
- `unique_copy(b, e, b2)` copy `[b,e)` to `[b2, b2+(e-b))` but
don't copy adjacent duplicates
- `merge(b, e, b2, e2, r)` merge two sorted sequence `[b2, e2)` and `[b, e)`
into `[r, r+(e-b)+(e2-b2))`
- `r=equal_range(b, e, v)` `r` is the subsequence of `[b,e)` with the value `v`
(basically a binary search for `v`)
- `equal(b, e, b2)` do all elements of `[b,e)` and `[b2, b2+(e-b))` compare
equal?


Iterators Make the Algorithms and Containers Independent

```
vector<int> v = {27, 210, 12, 47, 109, 83};  
int val = 83;  
vector<int>::iterator  
it_result = find(v.begin(), v.end(), val);
```

1. find accesses the first element in the vector
2. It compares that element to val
3. If this element matches, find returns the element iterator
4. Otherwise, find advances to the next element and repeats steps 2 and 3
5. find stops when it has reached the end of the sequence
6. If find gets to the end of the sequence, it returns v.end()
(in general its second iterator)

Don't use find with associative containers!!!!

Types of Algorithms - 1

1. Algorithms that read elements 
 - find, count, accumulate
2. Algorithms that write elements
 - fill, fill_n, copy, replace, replace_copy
3. Algorithms that rearrange the order of the elements
 - sort, partial_sort, is_sorted, unique, erase
4. Search algorithms
 - binary_search, equal_range

Types of Algorithms - 2

5. Non modifying algorithms
 - count, equal, mismatch, search
6. Modifying algorithms
 - fill, fill_n, move, transform, generate, swap
7. Numeric algorithms
 - iota, accumulate, partial_sum
8. Minimum and Maximum operations
 - max_element, min_element, minmax_element, next_permutation,...

Passing a Function to an Algorithm

- Many of the algorithms compare elements in the input sequence, using either the element type's `<` or `==` operator by default
- The library also defines versions of these algorithms that let us supply our own operation to use in place of the default operators
- These algorithms may take also callable objects or general functions, and call them on the elements in the input range

Passing a Function to an Algorithm

```
// comparison function to be used to sort by
// word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

Reading Algorithms: accumulate

The `accumulate` function, takes two `InputIterators` and an initial value to use in the summation. It then computes the sum of all of the elements contained in the range of iterators, plus the base value.

```
std::accumulate(v.rbegin(), v.rend(), init, bin_op)
```

`bin_op` is a binary operation callable object and is optional.

Reading Algorithms: `accumulate`

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr !=
values.end(); ++itr)
    total += *itr;
cout << "Average is: " << total / values.size() << endl;
```



```
cout << accumulate(values.begin(),
values.end(), 0.0)    / values.size() << endl;
```

Note

Ordinarily it is best to use `cbegin()` and `cend()` with algorithms that read, but do not write, the elements. However, if you plan to use the iterator returned by the algorithm to change an element's value, then you need to pass `begin()` and `end()`.

Reading Algorithms: equal

Compares the elements in the range denoted by InputIterators `[first1,last1)` with those in the range beginning at `first2`, and returns true if all of the elements in both ranges match.

```
bool equal (first1, last1, first2, pred);
```

The elements are compared using operator `==` or a binary callable object `comparator`, such as:

```
bool mypredicate (int i, int j) {  
    return (i==j);  
}
```


Reading Algorithms: equal

```
int myints[] = {20,40,60,80,100};  
std::vector<int> myvector(myints,myints+5);  
  
// using callable object comparison:  
if (std::equal (myvector.begin(), myvector.end(),  
               myints, mypredicate))  
    std::cout << "sequences are equal.\n";  
else  
    std::cout << "sequences differ.\n";
```

Algorithms that take a single iterator denoting a second sequence assume that the second sequence is at least as large at the first

Writing Algorithms: `fill_n`

Algorithms that write to a destination `OutputIterator` assume the destination is large enough to hold the number of elements being written.

```
vector<int> vec1;  
fill_n(vec1.begin(), vec1.size(), 0); //safe
```

```
vector<int> vec2;  
fill_n(vec2.begin(), 10, 0); // disaster
```

Writing Algorithms: using `back_inserter`

An insert iterator is an OutputIterator that adds elements to a container.

```
vector<int> vec3; //empty vector
// ok: back_inserter creates an insert
// iterator that adds elements to vec
// appends ten elements to vec3
fill_n(back_inserter(vec3), 10, 0);
```

```
vector<int> vec4; // empty vector
auto it = back_inserter(vec4); // assigning
                                // through it adds
                                // elements to vec4
```

```
*it = 42; // vec4 now has one element with value 42
```

Writing Algorithms: copy

Copies one container to another.

```
vector<int> v1 = {0,1,2,3,4,5,6,7,8,9};  
vector<int> v2(10);  
auto ret = copy(v1.begin(), v1.end(), v2.begin());
```

The value returned by `copy` is the (incremented) value of its destination iterator. That is, `ret` will point just past the last element copied into `v2`

Writing Algorithms: copy

Several algorithms provide so-called “copying” versions which compute new element values, but instead of putting them back into their input sequence, create a new sequence to contain the results.

```
// replace any element with the value 0 with 42  
replace(ilst.begin(), ilst.end(), 0, 42);
```

```
// use back_inserter to grow destination  
// as needed  
replace_copy(ilst.cbegin(), ilst.cend(),  
back_inserter(ivec), 0, 42);
```

Reordering Algorithms: `sort`

Takes two `RandomAccessIterators` denoting the range of elements to sort. In this call, we sort the entire vector.

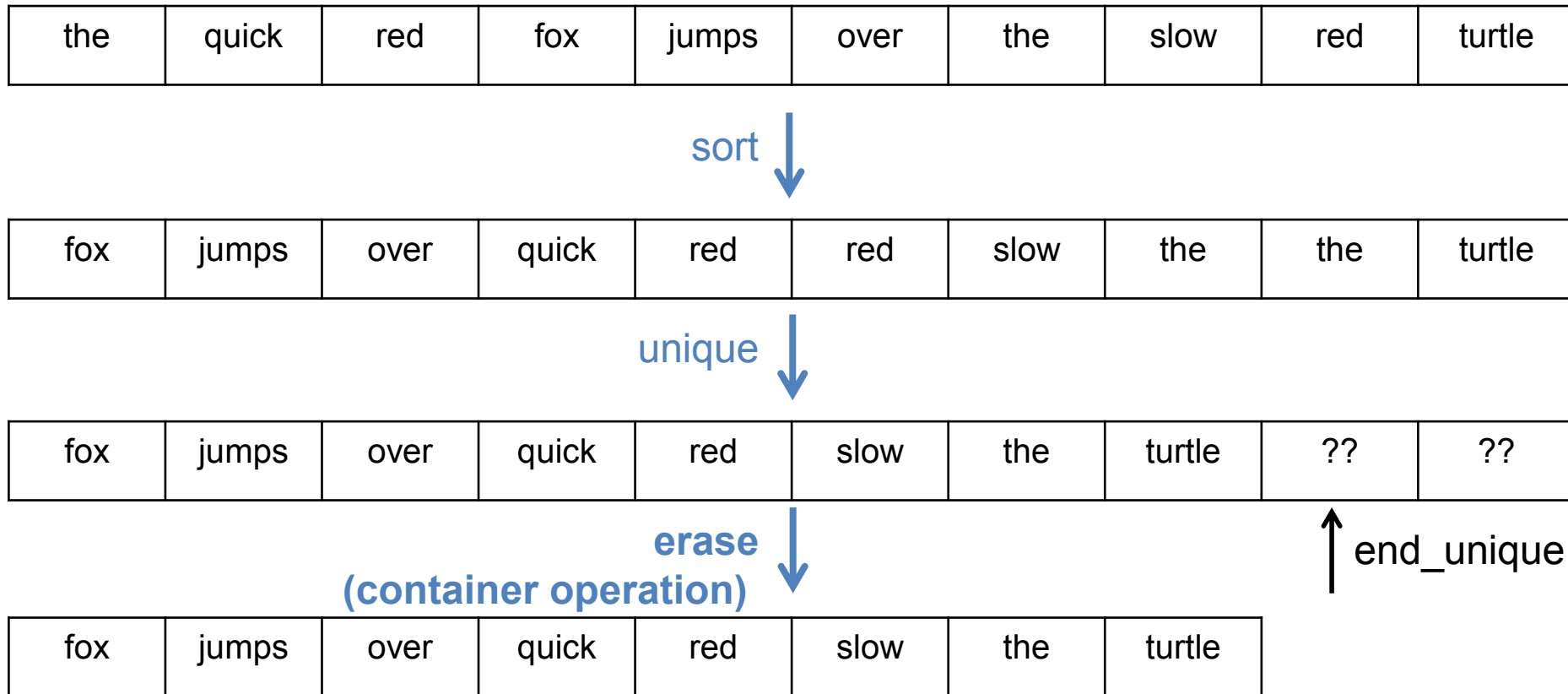
```
void elimDups(vector<string> &words)
{
    sort(words.begin(), words.end());
    auto end_unique = unique(words.begin(), words.end());
    words.erase(end_unique, words.end());
}
```

Reordering Algorithms: unique

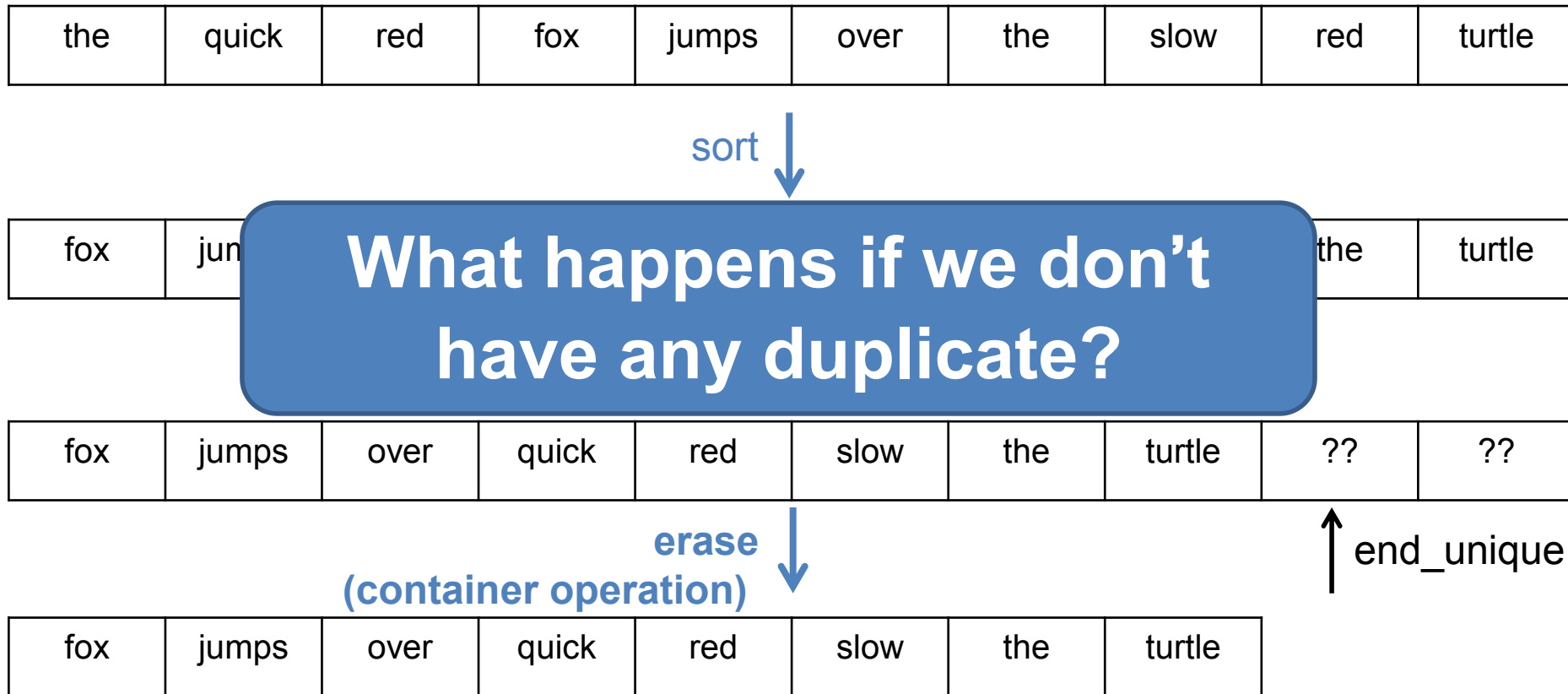
Rearranges the input range to “eliminate” adjacent duplicated entries, and returns an iterator that denotes the end of the range of the unique values.

```
void elimDups (vector<string> &words)
{
    sort (words.begin(), words.end());
    auto end_unique = unique (words.begin(), words.end());
    words.erase (end_unique, words.end());
}
```

elimDups execution



elimDups execution



Associative Containers and Algorithms

- We cannot pass associative container iterators to algorithms that write to or reorder container elements because keys are const
- The elements in the set types are const, and those in maps are pairs whose first element is const

Associative containers can be used with the algorithms that read elements, but this is not always a good idea!

Container Specific Algorithms

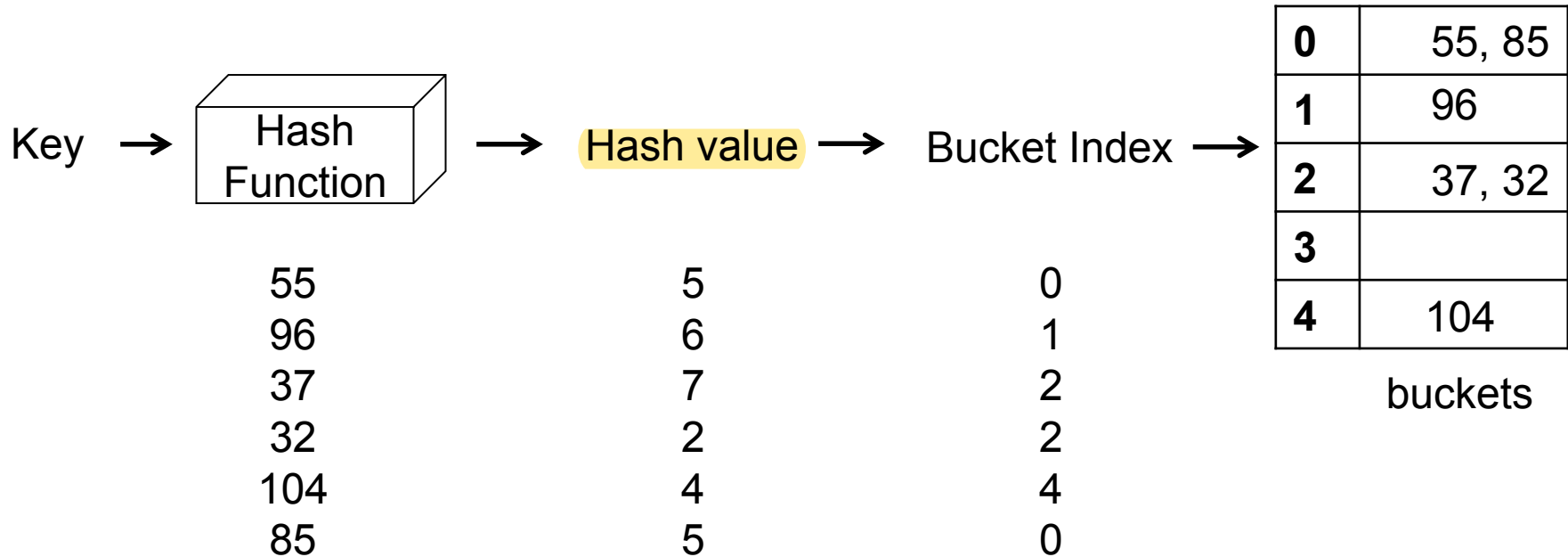
<code>lst.merge(lst2)</code>	Merge elements from <code>lst2</code> onto <code>lst</code> . Both <code>lst</code> and <code>lst2</code> must be sorted. After the merge <code>lst2</code> will be empty
<code>lst.merge(lst2, comp)</code>	Instead of <code><</code> operator, it uses a given comparison operations
<code>lst.remove(val)</code>	Calls <code>erase</code> for elements <code>==</code> to <code>val</code>
<code>lst.remove_if(pred)</code>	Calls <code>erase</code> for elements for which <code>pred</code> succeeds
<code>lst.reverse()</code>	Reverses the order of elements in <code>lst</code>
<code>lst.sort()</code> <code>lst.sort(comp)</code>	Sort the elements of <code>lst</code> using <code><</code> or <code>comp</code>
<code>lst.unique()</code> <code>lst.unique(pred)</code>	Call <code>erase</code> to remove consecutive copies of the same value using <code>==</code> or binary predicate

unordered_map & unordered_set

Unordered Associative Containers

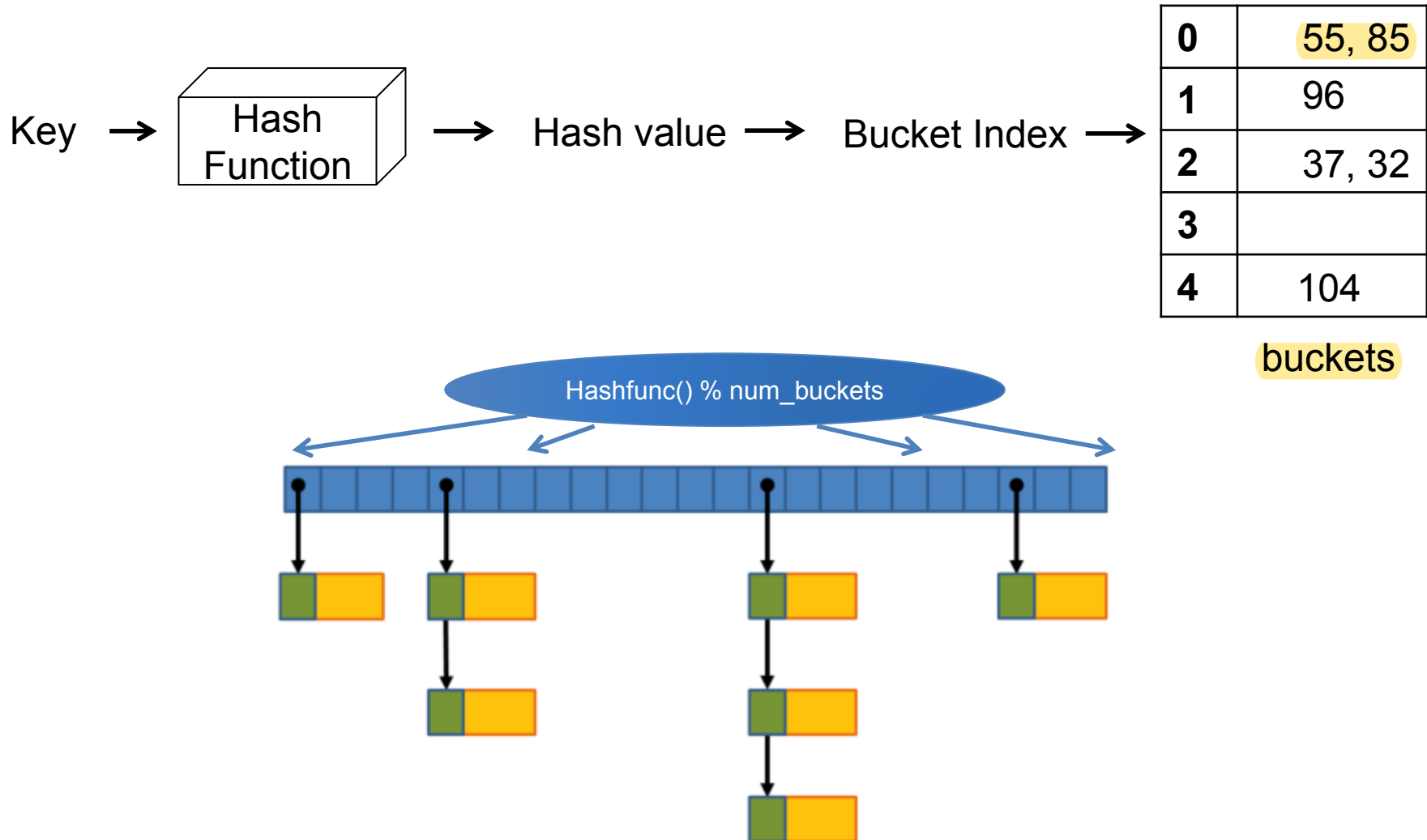
- Collection of buckets
 - Each bucket contains a variable number of items
- Use of a hash function to map elements to buckets
 - Given the item key, identify the proper bucket to store such item
 - All of the elements with a given hash value are stored in the same bucket
 - All the elements with the same key will be in the same bucket

Unordered Associative Containers



Different keys with the same bucket index are stored in the same bucket and originate a collision

Unordered Associative Containers



Requirements to use unordered containers

- To use an unordered container with a type `T` you need to define an equality operator by
 - overloading operator `==` or
 - specializing the standard function `equal_to<T>` or
 - defining a callable class
- You need also a hash function by
 - specializing `hash<T>` or
 - defining a callable class

Unordered Associative Containers

- The performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets
- Rather than using a comparison operation to organize their elements, these containers use a hash function and the key type's `==` operator
- Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve

Requirements on Key Type for Unordered Containers

```
struct hasher {  
    size_t operator() (const Sales_data &sd)  
    {  
        return hash<string>() (sd.isbn());  
    }  
};  
  
struct eqFunc{  
    bool operator() (const Sales_data &lhs,  
const Sales_data &rhs) {  
        return lhs.isbn() == rhs.isbn();  
    }  
};
```

We can use them to define an `unordered_set`.

Requirements on Key Type for Unordered Containers

```
using SD_unordered_set = unordered_set<Sales_data,  
                                     hasher, eqFunc>;
```

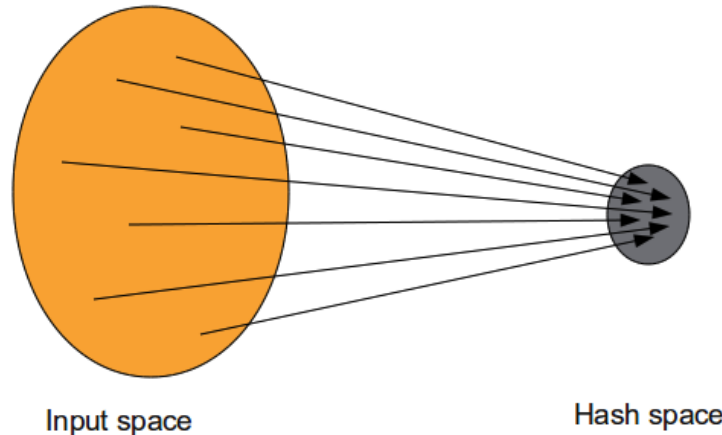
```
//argument is the bucket size  
SD_unordered_set bookstore(42);
```

std::hash<T>

- A unary function object type that takes an object of type `key_type` as argument and returns a unique value of type `size_t` based on it
- The unordered objects use the hash values returned by this function to organize their elements internally, speeding up the process of locating individual elements
- Aliased as member type, e.g., `unordered_map::hasher`
- The function **doesn't know which is the size of the unordered object**, simply transforms a `key_type` into a `size_t`
- For most common types an implementation of the hash function is already provided by the language

Define a Hash function

1. Accepts a single parameter of type `Key`
2. Returns a value of type `size_t` that represents the hash value of the parameter
3. $k1 == k2 \rightarrow \text{std::hash<Key>}() (k1) == \text{std::hash<Key>}() (k2)$
4. $k1 \neq k2, \text{std::hash<Key>}() (k1) == \text{std::hash<Key>}() (k2)$
should be rarely probable



Example of a hash function

```
struct S {  
    std::string first_name;  
    std::string last_name;  
};  
  
bool operator==(const S& lhs, const S& rhs)  
{  
    return lhs.first_name == rhs.first_name  
    && lhs.last_name == rhs.last_name;  
}
```

Example of a hash function

```
struct MyHash
{
    std::size_t operator()(S const& s) const {
        h = std::hash<std::string>()(s.first_name + s.last_name);
        return h;
    }
};
```

```
S obj = { "Hubert", "Farnsworth"};
std::cout << MyHash()(obj) << "(using MyHash)" << '\n';
```

void rehash(size_type n)

- A *rehash* is the reconstruction of the hash table:
 - All the elements in the container are rearranged according to their hash value into the new set of buckets
 - This may alter the order of iteration of elements within the container
- `rehash` sets the number of buckets in the container to `n` or more
- If `n` is greater than the `bucket_count`, a rehash is forced
 - The new `bucket_count` can either be equal or greater than `n`
- If `n` is lower than `bucket_count`, the function may have no effect on the `bucket_count` and may not force a rehash
- Rehashes are automatically performed by a container whenever its `load_factor` is going to surpass its `max_load_factor` in an operation
- `reserve` is similar but it expects the number of elements in the container as argument

Unordered Container Management Operations - Bucket Interface

Bucket Interface	
<code>c.bucket_count()</code>	Number of buckets in use.
<code>c.max_bucket_count()</code>	Largest number of buckets this container can hold.
<code>c.bucket_size(n)</code>	Number of elements in the n-th bucket.
<code>c.bucket(k)</code>	Bucket in which elements with key k would be found.

The number of buckets is increased when the size of the container increases to keep the average number of elements in each bucket under a certain value.

Unordered Container Management Operations - Bucket Iteration

Bucket Iteration	
<code>local_iterator</code>	Iterator type that can access elements in the bucket
<code>const_local_iterator</code>	const version of <code>local_iterator</code>
<code>c.begin(n)</code> , <code>c.end(n)</code>	Iterator to first, one past the last element of bucket <code>n</code> .
<code>c.cbegin(n)</code> , <code>c.cend(n)</code>	Returns <code>const_local_iterator</code>

Unordered Container Management Operations – Hash Policy

Hash Policy	
<code>c.load_factor()</code>	Average number of elements per bucket. Returns float.
<code>c.max_load_factor()</code>	Average bucket size that c tries to maintain. c adds buckets to keep <code>load_factor < max_load_factor</code> . Returns float.
<code>c.rehash(n)</code>	Reorganize storage so that <code>bucket_count ≥ n</code> and <code>bucket_count > size/max_load_factor</code> .
<code>c.reserve(n)</code>	Reorganize so that c can hold n elements without a rehash.

Readings

SEQUENTIAL CONTAINERS (Continue)

List & forward_list & strings

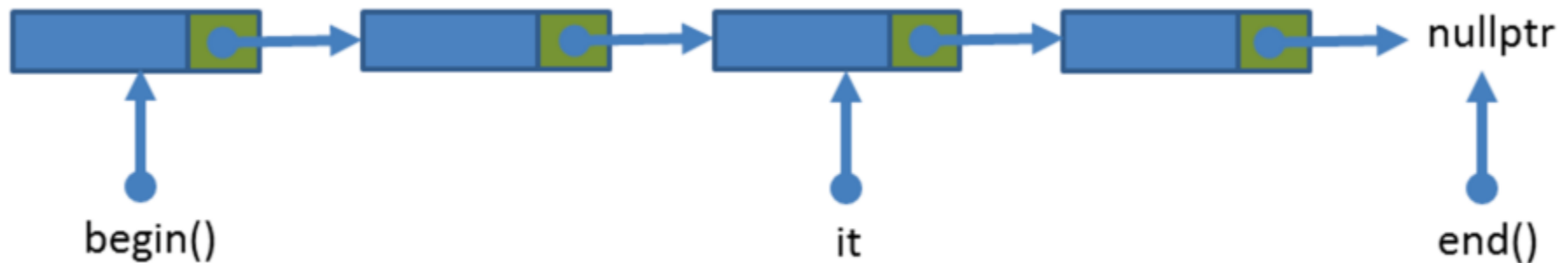
list <T>

Some of its containing methods are:

lst.push_front(t)	Inserts t at the beginning of lst
lst.push_back(t)	Adds t at the end of the lst
lst.pop_front()	Removes the first element in lst
lst.pop_back()	Removes the last element in lst
lst.emplace_front(args) lst.emplace_back(args) lst.emplace(p, args)	Constructs a new element from <i>args</i> and inserts it at the begin/end/before the element denoted by p in lst
lst.insert(p, t)	Inserts t before the element denoted by p in lst
lst.erase(p) lst.erase(b,e)	Removes from lst either a single element denoted by p or a range of elements [b,e)

forward_list <T>

It implements a singly-linked list. It provides only part of the functionalities of a list<T> but it is more efficient and less memory consuming.



`forward_list::end()` and `cend()` return an iterator to the element following the last element of the container. This element acts as a placeholder; attempting to access it results in undefined behavior.

Specialized forward_list operations

- To understand why forward_list has special versions of the operations to add and remove elements, consider what must happen when we remove an element from a singly linked list
- To add or remove an element, we need access to its predecessor in order to update that element's links
 - In a singly linked list there is no easy way to get to an element's predecessor
 - For this reason, the operations to add or remove elements in a forward_list operate by changing the element after the given element

forward_list Operations

<code>lst.before_begin()</code>	Returns the non existent element just before the beginning of the list. This iterator may not be dereferenced.
<code>lst.cbefore_begin()</code>	Returns a <code>const_iterator</code> .
<code>lst.insert_after(p,t)</code> <code>lst.insert_after(p,n,t)</code> <code>lst.insert_after(p,b,e)</code> <code>lst.insert_after(p,il)</code>	Inserts elements after the one denoted by iterator <code>p</code> . <code>t</code> is an object, <code>n</code> is a count and <code>b</code> and <code>e</code> are iterators denoting a range (they must not refer to <code>lst</code>) and il is a braced list . Returns an iterator to the last inserted element. If the range is empty returns <code>p</code> . undefined if <code>p</code> is off-the-end.
<code>emplace_after(p,args)</code>	Constructs a new element with <i>args</i> after <code>p</code> . returns an iterator to the new element.
<code>lst.erase_after(p)</code> <code>lst.erase_after(b,e)</code>	Removes the element after <code>p</code> or those in <code>[b,e)</code> . returns an iterator to the element after the one deleted or off-the-end iterator if there is no such element. undefined if <code>p</code> denotes the last element in <code>lst</code> or is off-the-end.

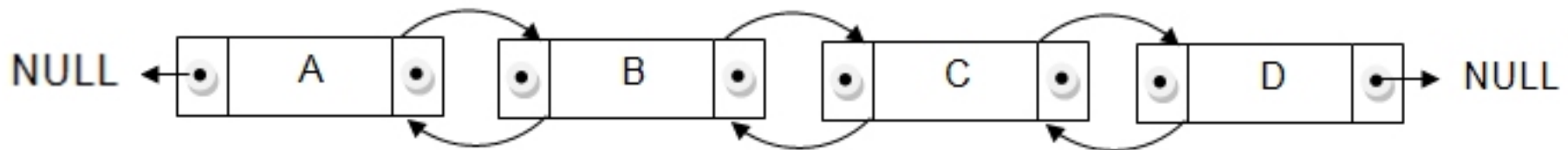
list<T> and forward_list <T>

The list and forward_list containers are designed to make it fast to **add or remove an element anywhere in the container**. In exchange, these types do not support random access to elements. We can access an element only by iterating through the container. Moreover, the memory overhead for these containers is often substantial, when compared to vector, deque, and array

- If your program has lots of small elements and space overhead matters, don't use list or forward_list
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list

list<T>

- It implements a doubly-linked list
- Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it



Doubly-Linked List

An example of `list<T>`

```
std::list<int> first;
//four ints with value 100
std::list<int> second(4,100);
//iterating through second
std::list<int> third (second.begin(), second.end());
//a copy of third
std::list<int> fourth (third);
// the iterator constructor can also be used to
// construct from arrays
List myints[] = {16,2,77,29};
std::list<int> fifth (myints, myints +
sizeof(myints) / sizeof(int));
```

list<T>

What is wrong with the following program? How might you correct it?

```
list<int> lst1;  
list<int>::iterator iter1 = lst1.begin(),  
iter2 = lst1.end();  
while (iter1 < iter2)  
    /* ... */
```

Example

```
auto prev = flst.before_begin();
// denotes element "off the start" of flst
auto curr = flst.begin();
while (curr != flst.end()) {
    if (*curr % 2) //if the number is odd
        curr = flst.erase_after(prev);
    else {
        prev = curr;
        // move the iterators to denote the next
        ++curr;
        //element and one before the next element
    }
}
```

Example

```
struct is_odd_struct
{
    bool operator() (const int& value)
        {return (value%2)==1; }
};

int main ()
{
    is_odd_struct is_odd_object;
    forward_list<int> mylist = {7,80,7,15,82};
    mylist.remove_if (is_odd_object); // 80 82
    return 0;
}
```

String Operations

The string type provides a number of additional operations beyond those common to the sequential containers.

String constructors	
<code>String s (cp , n)</code>	s is a copy of the first n characters in the array pointed by cp. That array must have at least n characters.
<code>String s (s2, pos2)</code>	s is a copy of s2 characters starting at position pos2. undefined if pos2 > s2.size().
<code>String s (s2, pos2, len2)</code>	s is a copy of len2 characters from s2 starting at the index pos2. undefined if pos2 > s2.size(). Regardless of the value of len2, copies at most s2.size()- pos2 characters.

Example

```
std::string s0 ("Initial string");  
std::string s1;  
std::string s2 (s0);  
std::string s3 (s0, 8, 3); // str  
std::string s4 ("Another character  
sequence", 12); //Another char  
std::string s5 (10, 'x');  
std::string s6 (s0.begin(), s0.begin()+7);  
//Initial
```

String Operations

The `substr` function raises an `out_of_range` error if the position exceeds the size of the string. If the position plus the count is greater than the size, the count is adjusted to copy only up to the end of the string.

Substring operation	
<code>s.substr(pos, n)</code>	Returns a string containing <code>n</code> characters from <code>s</code> , starting at <code>pos</code> . <code>pos</code> defaults to 0. <code>n</code> defaults to a value that causes the library to copy all the characters in <code>s</code> starting from <code>pos</code> .

Example

```
string s("hello world");  
string s2 = s.substr(0, 5); // s2 = hello  
string s3 = s.substr(6); // s3 = world  
string s4 = s.substr(6, 11); // s3 = world  
string s5 = s.substr(12); // out_of_range
```

String Operations

String Modification operation	
<code>s.insert(pos, args)</code>	Insert characters specific by <i>args</i> before <i>pos</i> . <i>pos</i> can be an index or iterator. Versions taking a index return a reference to s and those taking an iterator return an iterator to the first inserted character.
<code>s.erase(pos, len)</code>	Remove len characters starting at <i>pos</i> . If len is omitted, removes characters from pos to the end of s and returns a reference to s.
<code>s.assign(args)</code>	Replace characters in s according to <i>args</i> and returns a reference to s.
<code>s.append(args)</code>	Appends args to s. Returns a reference to s.
<code>s.replace(range, args)</code>	Removes range of characters from s and replace them by characters formed by args. Range is either an index and a length or a pair of iterators to s. Returns a reference to s.

Example

```
std::string str = "string1";  
std::string str2 = "string2";  
str.assign(2, '*');  
str.append(str2); /**string2  
str.append(str2, 3, 3); /**string2ing  
str.append("just 5 characters of this  
string", 5); /**string2ingjust
```

String Operations

String search operation	
<code>s.find(args)</code>	Find the first occurrence of <i>args</i> in <i>s</i> .
<code>s.rfind(args)</code>	Find the last occurrence of <i>args</i> in <i>s</i> .
<code>s.find_first_of(args)</code> <code>s.find_last_of(args)</code>	Find the first/last occurrence of any character from <i>args</i> in <i>s</i> .
<code>s.find_first_not_of(args)</code> <code>s.find_last_not_of(args)</code>	Find the first/last character in <i>s</i> that is not in <i>args</i> .

Where *args* has one of the following forms:

(*c*, *pos*)

(*s2*, *pos*)

(*cp*, *pos*)

(*cp*, *pos*, *n*)

Example

```
std::string str ("look for non-alphabetic  
characters...");
```

```
std::size_t found =  
    str.find_first_not_of("abcdefghijklmnopqrs  
tuvwxyz ");
```

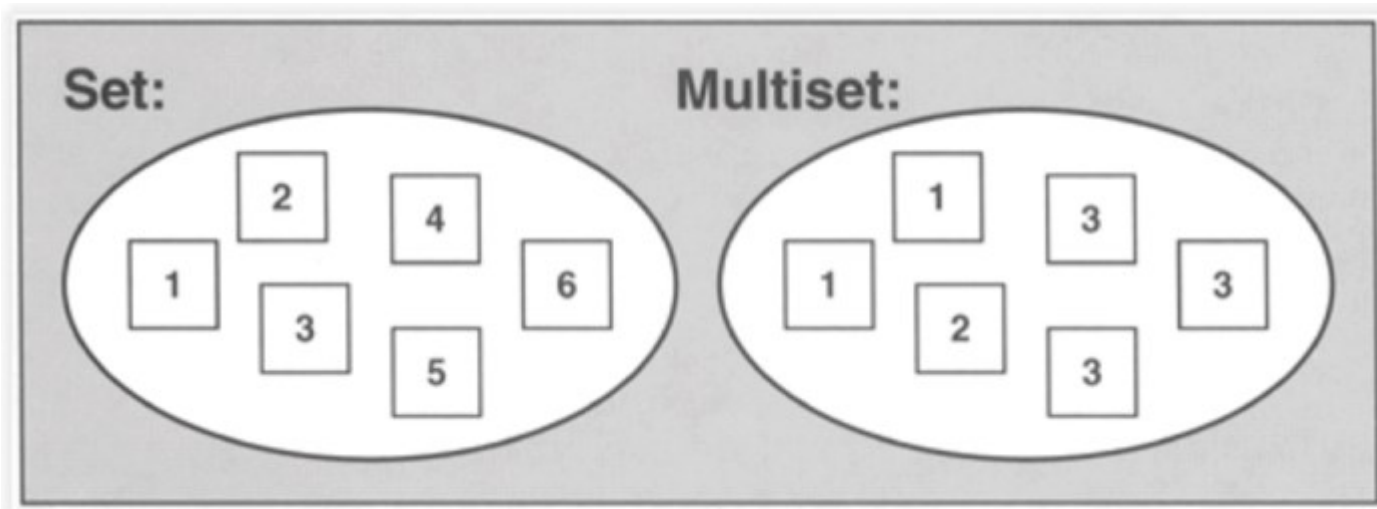
```
if (found!=std::string::npos)  
{  
    cout << str[found]<<"at" << found <<'\n';  
    //- at 12  
}
```

ASSOCIATIVE CONTAINERS (Continue)

List & forward_list & strings

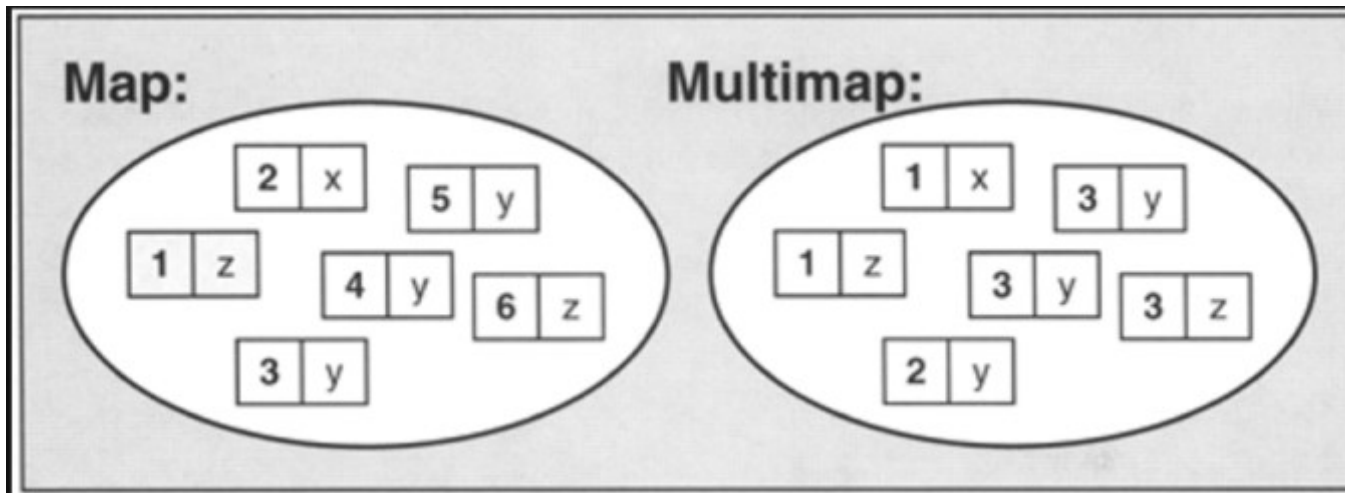
multiset and set

Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two is that multisets allow duplicates, whereas sets do not.



multimap and map

In multimap containers, despite, there can be several elements with the same key.



Finding Elements in a `multimap` or `multiset` using `upper/lower-bound`

If the key is in the container, `lower_bound` will refer to the first instance of that key and `upper_bound` will refer just after the last instance of the key. If not, then `lower_bound` and `upper_bound` will refer to the point at which the key can be inserted without disrupting the order.

```
for (auto beg = authors.lower_bound(search_item),  
     end = authors.upper_bound(search_item); beg != end; ++beg)  
    cout << beg->second << endl;
```

Finding Elements in a multimap or multiset- using `equal_range` function

`equal_range` takes a key and returns a pair of iterators. If the key is present, then the first iterator refers to the first instance of the key and the second iterator refers one past the last instance of the key. If no matching element is found, then both the first and second iterators refer to the position where this key can be inserted.

```
for (auto pos = authors.equal_range(search_item);  
pos.first != pos.second; ++pos.first)  
    cout << pos.first->second << endl;
```

Advanced Readings

Adding Elements

```
// more verbose way to count number of times
// each word occurs in the input

map<string, size_t> word_count;
string word;
while (cin >> word) {
    auto ret = word_count.insert({word, 1});
    if (!ret.second)
        ++ret.first->second;
}
```

Adding Elements

```
// more verbose
// each word occurs only once

map<string, size_t> m;
string word;
while (cin >> word)
{
    auto ret = m.insert({word, 1});
    if (!ret.second)
        ++ret.first->second;
}
```

- **ret** holds the value returned by insert, which is a pair
- **ret.first** is the first member of that pair, which is a map iterator referring to the element with the given key
- **ret.first->**dereferences that iterator to fetch that element. Elements in the map are also pairs
- **ret.first->second** is the value part of the map element pair.
- **++ret.first->second** increments that value.

REVISITING ITERATORS (Continue)

Revisiting Iterators

In addition to the iterators that are defined for each of the containers, the library defines several additional kinds of iterators in the *iterator* header.

- **Insert iterators:** These iterators are bound to a container and can be used to insert elements into the container
- **Stream iterators:** These iterators are bound to input or output streams and can be used to iterate through the associated IO stream
- **Reverse iterators:** These iterators move backward, rather than forward. The library containers, other than `forward_list`, have reverse iterators

Insert Iterators

When we assign a value through an insert iterator, the iterator calls a **container operation** to add an element at a specified position in the given container.

- `back_inserter` creates an iterator that uses `push_back`
- `front_inserter` creates an iterator that uses `push_front`
- `inserter` creates an iterator that uses `insert`. This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator

Insert Iterators

```
list<int> lst = {1,2,3,4};  
list<int> lst2, lst3; // empty lists
```

```
//lst2={4,3,2,1}  
copy(lst.cbegin(), lst.cend(),  
front_inserter(lst2));
```

```
//lst1={1,2,3,4}  
copy(lst.cbegin(), lst.cend(),inserter(lst3, lst3.begin()));
```

Stream iterators

Using a stream iterator, we can use the generic algorithms to read data from or write data to stream objects.

```
istream_iterator<int> in_iter(cin);  
    // read ints from cin  
istream_iterator<int> eof;  
// istream 'end' iterator  
while (in_iter != eof)  
    vec.push_back(*in_iter++);
```

Reverse Iterators

A reverse iterator is an iterator that traverses a container backward, from the last element toward the first.

```
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
```

```
for (auto r_iter = vec.crbegin();  
     r_iter != vec.crend();  
     ++r_iter)
```

```
    cout << *r_iter << endl;  
// prints 9, 8, 7,... 0
```

Splice Algorithm

This algorithm is particular to *list* and thus, does not have a generic version.

lst.splice(<i>args</i>) or flst.splice_after(<i>args</i>)	
(p,lst2)	p is an iterator to an element in lst or an iterator just before an element in flst. Moves all the element(s) from lst2 into lst just before p or into flst just after p. Removes the elements from lst2. lst2 must have the same type as lst or flst and may not be the same list.
(p,lst2,p2)	p2 is a valid iterator into lst2. Moves the element denoted by p2 into lst or moves the element just after p2 into flst. Lst2 can be the same list as lst or flst.
(p,lst2,b,e)	b and e must denote a valid range in lst2. moves the elements in the given range from lst2. lst2 and lst (or flst) can be the same list but p must not denote an element in the given range.

References

- Lippman Chapter 10,11
- <http://www.cplusplus.com/reference/stl/>
- <http://www.learncpp.com/cpp-tutorial>

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming